UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Rogério Bodemüller Junior

MECANISMOS DE PREVISÃO DE PERDA DE DEADLINE PARA TRATADORES DE EVENTOS RTSJ

Florianópolis

2014

Rogério Bodemüller Junior

MECANISMOS DE PREVISÃO DE PERDA DE DEADLINE PARA TRATADORES DE EVENTOS RTSJ

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação para a obtenção do Grau de Mestre.
Orientadora: Prof.ª Patricia Della Méa Plentz, Dr.ª

Florianópolis

2014

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Bodemüller Junior, Rogério
Mecanismos de previsão de perda de deadline para tratadores de eventos RTSJ / Rogério Bodemüller Junior ; orientadora, Patricia Della Méa Plentz - Florianópolis, SC, 2014.
101 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da Computação. I. Plentz, Patricia Della Méa. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Rogério Bodemüller Junior

MECANISMOS DE PREVISÃO DE PERDA DE DEADLINE PARA TRATADORES DE EVENTOS RTSJ

Esta Dissertação foi julgada aprovada para a obtenção do Título de "Mestre", e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 03 de Julho 2014.

Prof. Ronaldo dos Santos Mello, Dr.
Coordenador do Curso

Banca Examinadora:

Prof.ª Patricia Della Méa Plentz, Dr.ª
Orientadora

Prof. Rômulo Silva de Oliveira, Dr.
Universidade Federal de Santa Catarina

Prof. Mario Antonio Ribeiro Dantas, Dr.
Universidade Federal de Santa Catarina

Prof.ª Luciana de Oliveira Rech, Dr.ª
Universidade Federal de Santa Catarina

Dedico este trabalho à todo aquele que é movido a desafios. Conheça seus limites mas nunca os aceite!

AGRADECIMENTOS

Agradeço primeiramente a Deus, que me forneceu força para executar o presente trabalho, e à minha amada esposa, Priscila Costa Schmidt Bodemüller, pelo seu apoio incondicional, companheirismo, amor e amizade, além da compreensão nos momentos de ausência física e espiritual.

Com carinho, agradeço aos meus pais, Rogério Bodemüller e Marilena Fischer Bodemüller, pelo suporte que me propiciaram para chegar neste ponto, pelo imenso carinho que eles sempre tiveram comigo e pelos ensinamentos que eu levarei para o resto da minha vida. E ao meu irmão, Richard José Bodemüller, pelo seu companheirismo, mesmo que do nosso jeito.

Agradeço à minha orientadora, Patricia Della Méa Plentz, pelo auxílio no desenvolvimento deste trabalho, compreensão e dedicação, mesmo tendo passado por uma mágica gravidez durante a orientação.

Agradeço à Katiana, secretária do PPGCC, por sempre ter me ajudado em tudo que precisei. Ao professor Ricardo Felipe Custódio agradeço pelos desafios e oportunidades proporcionados quando estava trabalhando no LabSEC anos atrás e principalmente pelas lições deixadas. Agradeço também aos professores e colegas de estudo que tive a oportunidade de conhecer e trabalhar.

Aos colegas e amigos pessoais agradeço pelo incentivo e horas de diversão que me fizeram rir e relaxar. Aos Mestres, alunos e companheiros de Pa-Kua, obrigado pelos treinos, tanto os mais leves quanto os pesados, que me possibilitaram desestressar.

Por fim, agradeço a todas as pessoas que me ajudaram de forma direta ou indireta na realização deste trabalho, à UFSC e à Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina (FAPESC) pelo apoio.

Estudar, praticar e aperfeiçoar-se sempre.

(Gishin Funakoshi)

RESUMO

Estratégias para estimar a probabilidade de deadlines firmes serem alcançados são fundamentais porque permitem a realização de ações corretivas para a melhoria do desempenho do sistema. Este tipo de estratégia permite a avaliação de sistemas de tempo real que estão em funcionamento, principalmente quando há alguma mudança quanto ao projeto inicial, ou mesmo avaliar novos projetos para analisar se as restrições temporais foram definidas adequadamente. Nesta dissertação são apresentados dois mecanismos de previsão de perda de deadline em sistemas monoprocessados e multiprocessados de tempo real firme. O Mecanismo de Previsão de Perda de Deadline Baseado na Folga (MBF) utiliza dados sobre o comportamento das tarefas (deadline, tempo de computação e o tempo de espera na fila de prontos do processador) para calcular a folga e determinar se o deadline pode ser cumprido. O Mecanismo de Previsão de Perda de Deadline Baseado no Histórico (MBH) utiliza regressão linear e relaciona dados de um histórico de execuções passadas, que possui o tamanho da fila de prontos do processador e seu respectivo tempo de resposta, com o tamanho atual da fila de prontos do processador para calcular o tempo de resposta previsto da thread e depois definir a probabilidade dela cumprir seu deadline. Será apresentado um modelo de tarefas para aplicações não críticas em um sistema de tempo real firme que caracteriza uma aplicação real utilizada nos testes, o *cruise control*. Estes testes foram feitos utilizando uma implementação em Java RTS desta aplicação em um ambiente não especialista, próximo de um ambiente de tempo real comum, com várias configurações, buscando abranger uma grande gama de cenários. Após os testes, a qualidade das previsões é avaliada utilizando as Métricas Taxa Relativa de Erro e Taxa de Previsões Corretas. Os resultados demonstram que ambos os mecanismos trazem bons resultados em ambientes com cargas baixas, médias e altas, sendo o MBF um excelente previsor para sistemas monoprocessados e o MBH mais adequado aos sistemas multiprocessados.

Palavras-chave: Tempo Real. Previsão de Perda de Deadline. Java. RTSJ. Java RTS.

ABSTRACT

Strategies to estimate the probability of firm deadlines be achieved are essential because they allow the use of corrective actions to improve system performance. This type of strategy allows the evaluation of real-time systems that are in operation, especially when there is any change on the initial design, or evaluate new projects to analyze whether the temporal constraints were appropriately settled. In this dissertation, two deadline missing prediction mechanisms for firm real-time uniprocessor and multiprocessor systems are presented. The Deadline Missing Prediction Mechanism Based on Slack (MBF) uses data of tasks's behavior (deadline, computation time and the waiting time in the processor ready queue) to calculate the slack and determine whether the deadline can be met. The Deadline Missing Prediction Mechanism Based on Historical Data (MBH) uses linear regression and associates data from a past execution's historical, which is the size of the processor ready queue and its associated response time, with the current size of processor ready queue to calculate the predicted response time of the thread and then define the probability of meeting its deadline. A model of tasks for non-critical applications in a firm real-time system which characterizes a real application, similar to the cruise control, will be used in the tests. These tests were done using an implementation in Java RTS applied to a non-specialist environment as a common real-time environment with various configurations scenarios. The quality of the forecasts is evaluated using the metrics Relative Error Rate and Correct Prediction Rate. The results indicate that both mechanisms improve the performance in environments with high, medium and low system load whereas the MBF being an adequate predictor for uniprocessor systems and the MBH best suited to multiprocessor systems.

Keywords: *Real-Time. Deadline Missing Prediction. Java. RTSJ. Java RTS.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

LISTA DE SÍMBOLOS

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Aplicações de sistemas de tempo real são onipresentes e estão crescendo conforme a maturidade do hardware e do software cresce. Eles estão aparecendo como parte de infraestruturas comerciais, governamentais, médicas, educacionais e culturais (SHAW, 2001), tais como os sistemas de controle de tráfego aéreo, sistemas de comunicação e sistemas de automação predial. Um sistema de tempo real é definido como um sistema que deve satisfazer restrições temporais ou arriscar consequências que, inclusive, podem ser graves (LAPLANTE, 2006). Nesses tipos de sistema a correção temporal é tão importante quanto à correção lógica.

No desenvolvimento de aplicações de tempo real, as organizações tem enfrentado um dilema entre utilizar uma linguagem de programação popular ou uma especialista (BRUNO; BOLLELLA, 2009). Uma solução é a utilização de um padrão aberto baseado em uma plataforma que formalmente lida com desafios impostos por sistemas de tempo real usando uma linguagem de programação convencional, como o *Java Real-Time System* (Java RTS), desenvolvido pela antiga Sun, que implementa a *Real-Time Specification for Java* (RTSJ) (JCM, 2012).

A especificação RTSJ é uma extensão da plataforma padrão Java de forma que as restrições impostas por sistemas de tempo real são respeitadas, como previsibilidade e determinismo (DIBBLE, 2002). A RTSJ acrescenta ao Java padrão as seguintes características (BRUNO; BOLLELLA, 2009): adiciona threads tempo real, implementa um gerenciador de eventos assíncronos e um mecanismo para transferência assíncrona de controle entre threads, permite a execução de código sem influência do coletor de lixo e controla a localização de objetos e o acesso à memória em endereços físicos.

As restrições temporais são impostas pelo ambiente da aplicação, sendo uma delas denominada deadline. Segundo Liu (2000), um deadline pode ser crítico (*hard*), não crítico (*soft*) ou firme (*firm*). Um deadline é crítico se o descumprimento da restrição temporal é considerado uma falha fatal, ou seja, as consequências excedem em muito os benefícios do sistema em operação normal. Em contraste, o descumprimento de um deadline não crítico causa apenas uma queda no desempenho do sistema e não danos graves, em outras palavras, as consequências são da mesma ordem de grandeza que os benefícios normais

do sistema. Um deadline é firme se o descumprimento da restrição temporal não é considerado uma falha fatal, mas uma conclusão tardia não traz nenhum benefício para o sistema.

O contexto deste trabalho refere-se aos sistemas de tempo real firme monoprocessados e multiprocessados. Sistemas nesta categoria são constituídos por uma grande quantidade de tarefas aperiódicas, que representam as ligações a outros sistemas e sensores. Neste caso, informações precisas sobre a provável perda de um deadline podem beneficiar o sistema como um todo. O uso de mecanismos de previsão de perda de deadline é uma forma de melhorar o desempenho do sistema. Ações corretivas podem ser executadas para evitar a perda de deadline, ou evitar o desperdício no uso de recursos computacionais por tarefas que provavelmente não irão cumprir seu deadline e, consequentemente, não irão trazer benefícios ao sistema.

1.2 OBJETIVOS

O objetivo deste trabalho é apresentar uma estratégia para estimar a probabilidade de deadlines serem alcançados em sistemas monoprocessados e multiprocessados de tempo real firme, utilizando RTSJ, que leve em consideração dados atuais da thread em execução e um histórico de suas execuções passadas. Ele foi inspirado em modelos descritos em (PLENTZ; MONTEZ; OLIVEIRA, 2008b)(PLENTZ; MONTEZ; OLIVEIRA, 2011a)(PLENTZ; MONTEZ; OLIVEIRA, 2011b). A novidade deste trabalho é propor uma previsão para tratadores de eventos assíncronos, que são componentes do sistema que possuem um comportamento aperiódico porque dependem de entradas de outros sistemas e sensores.

Para atender o objetivo geral desta dissertação, os seguintes objetivos específicos foram definidos:

- Definir quais informações podem ser utilizadas para as previsões.
- Desenvolver um modelo de tarefas firmes para aplicações implementadas em um sistema monoprocessado.
- Desenvolver um modelo de tarefas firmes para aplicações implementadas em um sistema multiprocessado.
- Realizar testes em um ambiente que simule uma aplicação real.
- Avaliar a qualidade das previsões realizadas utilizando métricas, considerando diferentes contextos de execução.

1.3 MÉTODO DE PESQUISA

O trabalho será desenvolvido por meio da realização de uma revisão bibliográfica sobre previsores de perda de deadline que fazem uso de dados históricos. A partir da análise dos mecanismos apresentados na literatura, será proposto um previsor para sistemas de tempo real firme. A validação deste mecanismo será feita pela implementação e testes em ambiente que simule uma aplicação real.

A seguir, são apresentados os passos seguidos para a realização deste trabalho:

- Definir quais dados históricos serão usados pelo mecanismo de previsão.
- Propor mecanismos de previsão.
- Buscar uma implementação da RTSJ.
- Definir um ambiente de tempo real.
- Especificar um modelo de tarefas que simule uma aplicação real e que consiga se comportar de forma genérica, abrangendo diversas configurações das variáveis de controle do ambiente de teste.
- Realizar a implementação do ambiente de teste, dos mecanismos de previsão e das métricas para avaliação.
- Executar testes, ajustes no sistema e análise dos resultados.

1.4 LIMITAÇÕES DO TRABALHO

Este trabalho está focado na proposta de um mecanismo de previsão de perda de deadline em sistemas de tempo real firme utilizando tratadores de evento da RTSJ que leve em consideração dados históricos. Não farão parte do escopo do trabalho os demais tipos de sistemas de tempo real (tais como sistemas autônomos, embutidos ou distribuídos), sistemas de tempo real críticos, técnicas de inteligência artificial para o cálculo da relação entre os dados históricos e também a tomada de decisão após a realização da previsão. Essas e outras são perspectivas de trabalhos futuros, apresentadas no capítulo conclusivo.

1.5 ESTRUTURA DO TRABALHO

Esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta conceitos que constituem os pilares deste trabalho (sistemas de tempo real, linguagens de programação para tempo real e RTSJ). No Caítulo 3 é apresentado uma revisão bibliográfica sobre mecanismos de previsão. O Capítulo 4 descreve os dois mecanismos de previsão de perda de deadline para tratadores de eventos RTSJ propostos, suas características, tais como o algoritmo de utilização e o overhead causado, e duas métricas para avaliação de qualidade. Para poder avaliar estes mecanismos, eles foram adicionados em uma implementação de um estudo de caso. Este estudo de caso, os testes e os resultados, assim como o ambiente dos testes, o fluxo de execução e características da implementação e das cargas, são apresentados no Capítulo 5. Finalmente, no Capítulo 6 são apresentadas as conclusões e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta conceitos que constituem os pilares deste trabalho: sistemas de tempo real, demonstrado na Seção 2.1, linguagens de programação para tempo real na Seção 2.2 e o RTSJ, descrito na Seção 2.3.

2.1 SISTEMA DE TEMPO REAL

Aplicações de sistemas de tempo real são onipresentes e estão proliferando, aparecendo como parte de nossas infraestruturas comerciais, governamentais, médicas, educacionais e culturais (SHAW, 2001), tais como os sistemas de controle de tráfego aéreo, sistemas de comunicação e sistemas de automação predial.

Um dos primeiros livros relacionado a sistemas de tempo real foi publicado em 1967 por James Martin (MARTIN, 1967) e definiu tal termo como: aquele que controla um ambiente recebendo dados, processando estes dados, e executando uma ação ou retornando resultados suficientemente rápidos para afetar o funcionamento do ambiente naquele momento. Com mais de quatro décadas esta definição ainda encontra-se consistente, entretanto, o termo foi frequentemente utilizado de forma incorreta porque se associava o termo "tempo real" a um sistema online ou ainda a sistemas com velocidade de processamento. Atualmente a definição mais utilizada é: um sistema que deve satisfazer restrições temporais ou arriscar consequências que, inclusive, podem ser graves (LAPLANTE, 2006).

2.1.1 Conceitos Básicos

O elemento básico de execução em um sistema moderno de computação é chamado de tarefa, ou *thread*, a qual pode receber dados, executar um algoritmo específico e gerar algum tipo de saída. Em sistemas de tempo real estas tarefas devem estar corretas tanto no aspecto lógico, gerando um resultado correto, quanto sob o aspecto temporal, produzindo o resultado dentro de um prazo satisfatório. Um resultado que ocorra além do prazo especificado pode ser sem utilidade ou até representar uma ameaça (FARINES; FRAGA; OLIVEIRA, 2000). Esta restrição temporal na qual uma tarefa deve concluir sua execução, imposta

pelo ambiente da aplicação, é chamada de deadline.

Segundo Liu (2000), um deadline pode ser crítico (*hard*), não crítico (*soft*) ou firme (*firm*):

- Deadline crítico: se o descumprimento da sua restrição temporal é considerado uma falha fatal, ou seja, as consequências excedem em muito os benefícios do sistema em operação normal.
- Deadline não crítico: se o descumprimento causa apenas uma queda no desempenho do sistema e não danos graves ao mesmo. Em outras palavras, as consequências são da mesma ordem de grandeza que os benefícios normais do sistema.
- Deadline firme: se o descumprimento de sua restrição temporal não é considerado uma falha fatal, mas a sua conclusão tardia não traz nenhum benefício para o sistema.

Um sistema composto por tarefas com deadline críticos é dito sistema de tempo real crítico e tem como exemplo equipamentos médicos que dão suporte à vida. Por outro lado, um exemplo de sistema de tempo real não crítico é o aparelho de micro-ondas.

Além do deadline, outras restrições temporais podem ser definidas para uma tarefa de tempo real. Buttazzo (2004) cita as seguintes:

- Tempo de chegada: é o tempo no qual uma tarefa se torna pronta para execução;
- Tempo de liberação: é o instante no qual uma tarefa é inserida na fila de pronto;
- Tempo de computação: tempo necessário para o processador executar a tarefa;
- Tempo de início: tempo no qual uma tarefa inicia sua execução;
- Tempo de resposta: tempo no qual uma tarefa conclui sua execução;
- Importância: representa a importância relativa da tarefa com relação às outras do sistema;
- Latência: representa o atraso no tempo de resposta de uma tarefa;
- *Jitter*: máxima variação dos tempos de liberação das instâncias da tarefa;

- Folga: é o tempo máximo que uma tarefa pode ser atrasada na sua ativação para que o tempo de resposta seja menor ou igual ao deadline da tarefa.

As tarefas também podem ser classificadas quanto à periodicidade de suas ativações, conforme abaixo:

- Tarefas periódicas: as ativações do processamento das tarefas ocorrem sempre em um mesmo intervalo de tempo (período);

- Tarefas aperiódicas: as ativações da tarefa são desencadeadas por eventos internos ou externos, definindo uma característica aleatória nas ativações;

  – Tarefa esporádica: um subgrupo das tarefas aperiódicas que exige um intervalo mínimo de tempo (maior que zero) entre duas ativações sucessivas (OLIVEIRA, 1997).

2.1.2 Escalonamento de Tarefas de Tempo Real

O escalonador é o responsável por decidir qual tarefa irá executar quando houver mais de uma pronta para execução (CRUZ; LIMA, 2006). Ele é o componente básico para garantia de bom funcionamento em sistemas computacionais, pois pode otimizar o tempo de resposta de um processo.

O diagrama de estados da Figura 1 apresenta a forma simplificada dos possíveis estados de um processo dentro do sistema, descritos abaixo:

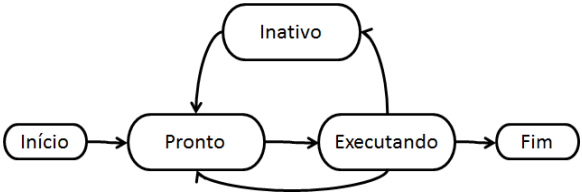

Figura 1: Estados de um processo

- Executando: tarefa está executando na *Central Processing Unit* (CPU).

- Inativo: tarefa indisponível para execução até que ocorra um evento externo.
- Pronto: tarefa disponível para execução e aguardando finalização de outro processo.

O escalonador executa uma política de escalonamento que ordena um conjunto de tarefas para execução no processador. Se as escalas produzidas forem viáveis, garantem o cumprimento das restrições temporais das tarefas tempo real.

Pode-se classificar uma política de escalonamento de acordo com:

- Preempção:
  - Preemptiva: a política pode interromper a execução de uma tarefa, a qualquer tempo, atribuindo o processador para outra tarefa de maior prioridade, ou maior importância.
  - Não-Preemptiva: nesta, a tarefa conclui sua execução sem ser interrompida por qualquer outra.
- Carga computacional:
  - Estática: a política baseia as decisões de escalonamento em parâmetros fixos que são atribuídos para as tarefas antes de sua execução. Nesse caso, todas as tarefas são periódicas ou esporádicas.
  - Dinâmica: a política utiliza parâmetros dinâmicos, que podem mudar durante a evolução do sistema. Essa política ocorre se pelo menos uma tarefa é aperiódica.
- Ativação das tarefas:
  - Offline: esta política é executada antes da ativação das tarefas no sistema. O escalonador, em tempo de execução, consulta a tabela que foi gerada anteriormente e cumpre a escala nela definida.
  - Online: nessa política as decisões de escalonamento ocorrem em tempo de execução (escalonamento não é pré-calculado), sempre que uma tarefa chega no sistema ou quando uma tarefa conclui sua execução.

Na política de escalonamento online, as tarefas são ordenadas segundo suas prioridades, onde a tarefa de maior prioridade é escalonada

e executada. As políticas de escalonamento se diferenciam na maneira como as prioridades são atribuídas a cada tarefa e são classificados como:

- Prioridade fixa: quando todas as ativações de uma mesma tarefa possuem a mesma prioridade, tal como o *Rate Monotonic* (RM) (LIU, 2000); e
- Prioridade dinâmica: quando cada ativação pode possuir uma prioridade diferente, como o *Earliest Deadline First* (EDF) (BUTTAZZO, 2004).

2.2 LINGUAGENS DE PROGRAMAÇÃO PARA TEMPO REAL

As aplicações de tempo real possuem exigências que as linguagens de programação tradicionais não cumprem. Muitas linguagens foram especialmente projetadas ou modificadas para tratar estes requisitos singulares. Um conjunto de critérios surgiu para comparar suas facilidades e adequação para o papel, e cada uma oferece seus próprios pontos fortes e fracos. Uma linguagem de programação de sistemas tempo real deve facilitar o acesso e controle do tempo, o controle de concorrência, o tratamento de exceções e a previsibilidade (SHAW, 2001). Alguns exemplos destas linguagens são: ADA, Esterel, OCCAM, MACH, Eiffel, MARUTI, Real-Time Euclid, Real-Time CORBA, PEARL, Real-Time C, Real-Time C++ e Java com extensão de tempo real.

Entre tantas opções, escolher uma linguagem adequada para aplicações em tempo real é complexo e demorado. Na maioria das circunstâncias a linguagem adotada para o projeto nunca está realmente em questão: utiliza-se a que está disponível, ou aquela que a equipe tem domínio (WILLIAMS, 2005). Ter tempo para pesquisar uma nova linguagem, ou mesmo um novo compilador, é um luxo raro que poucas equipes de projeto podem pagar.

Segundo Laplante (2006) as linguagens podem ser divididas em três tipos: assembly, procedural e orientada a objetos. Embora faltando a maioria das características de linguagens de alto nível, a linguagem assembly tem certas vantagens para uso em sistemas de tempo real, na medida em que fornece um controle mais direto e preciso do hardware do computador. Infelizmente ela não é estruturada, têm propriedades de abstração limitadas, a codificação em linguagem assembly é geralmente difícil de aprender, além do código resultante não ser portável.

Alguns exemplos de linguagens procedurais são C (ou Real-Time C), Fortran, Basic, Ada 2005, Modula-2, e Pascal. Estas são linguagens em que a ação do programa é definida por uma série de operações executadas em sequência e que podem ser agrupadas em *procedures* (módulos). Algumas funcionalidades destas linguagens que são de interesse em sistemas de tempo real são: mecanismos versáteis de passagem de parâmetros, tipagem forte, alocação dinâmica de memória, tipagem de dados abstratos, tratamento de exceções e a modularidade.

A versão original do Ada foi o resultado de uma competição internacional para escolher a linguagem de programação que seria utilizada pelo Departamento de Defesa dos Estados Unidos. É uma linguagem imperativa baseada em Pascal, muito extensa, fortemente tipada e com a capacidade de aceitar projetos orientados a objetos. Ada atrai entusiastas, mas não foi adotado pelo setor não-militar, em parte devido ao custo de compiladores (WILLIAMS, 2005). Ada 95 foi a primeira linguagem de programação orientada ao objeto padronizada internacionalmente e seu nome Ada é homenagem a condessa de Lovelace, Augusta Ada Byron, que trabalhou com Charles Babage, sendo considerada a primeira programadora de computadores da história.

Os benefícios das técnicas orientadas a objetos são bem conhecidos, tais como o aumento da eficiência do programador, a confiabilidade e o potencial de reutilização de código. Alguns exemplos incluem Smalltalk, C++, Java, C#, Eiffel, e Ada 2005 quando então usado. Formalmente, linguagens de programação orientadas a objeto são aqueles que suportam abstração de dados, herança e polimorfismo (LAPLANTE, 2006).

O Java oferece uma enorme comunidade de desenvolvedores, várias ferramentas e recursos e possui um histórico comprovado, mas é não-determinístico, introduzindo uma ilimitada quantidade de risco em aplicações que necessitam de garantia de tempo real. As outras linguagens para tempo real são determinísticas e previsíveis, garantindo que os requisitos temporais serão conhecidos, mas possuem ambientes complexos, comunidades de desenvolvimento pequenas, poucos recursos e ferramentas, além de não interagirem bem com aplicações atuais. A solução ideal para o desenvolvimento de aplicações de tempo real é a utilização de um padrão aberto que implementa os conceitos de tempo real usando uma linguagem de programação convencional e popular como o Java e uma extensão para tempo real.

O Java convencional não é adequado para o desenvolvimento de sistemas tempo real pois, dentre outros motivos, as Máquinas Virtuais Java (*Java Virtual Machine* - JVM) existentes não foram projetadas

levando-se em consideração previsibilidade e determinismo. Por exemplo, tanto o coletor de lixo quanto o compilador *Just-in-Time* (JIT) são não-determinísticos, acarretando uma fonte de latência e *jitter* sem limites. E, mais importante, o *Java Language Specification* (JLS) não fornece nenhuma garantia de execução de tarefas prioritárias, impossibilitando a construção de uma aplicação de tempo real (BRUNO; BOLLELLA, 2009).

Para resolver estes dois principais grupos de problemas (gerenciamento de memória e escalonamento de threads), em 1998 o *Java Specification Request* 001 (JSR-001) (PROCESS, 2012a) foi proposto com o objetivo de definir a especificação de Java para tempo real denominada RTSJ. Em 2000 esse grupo liberou a primeira especificação, que foi aprimorada e aceita em 2002 pelo *Java Community Process* (JCP), lançando a RTSJ 1.0. Em 2005 a revisão 1.0.1 da especificação foi publicada. No ano seguinte, trabalhando na RTSJ 1.1, deu-se o início do JSR-282 (PROCESS, 2012b). E desde 2006 a RTSJ continua em estágio de desenvolvimento com a revisão 1.0.2 (JCM, 2012).

2.3 *REAL-TIME SPECIFICATION FOR JAVA*

A RTSJ define uma API para a linguagem Java que permite a criação, verificação, análise, execução e gerenciamento de threads tempo real, procurando satisfazer seus requisitos temporais (PLENTZ et al., 2004). Alguns dos principais objetivos da RTSJ são (BRUNO; BOLLELLA, 2009):

- Compatibilidade: códigos Java externos devem funcionar corretamente na implementação RTSJ.
- Sintaxe Java: não deve haver adição ou mudança de termos da linguagem Java na RTSJ.
- Portabilidade: deve-se respeitar a política WORA (*write-once-run-anywhere*), que garante que a aplicação funcione em qualquer lugar.
- Flexibilidade: RTSJ deve permitir flexibilidade nas decisões como, por exemplo, garantir o funcionamento de uma implementação de escalonador diferente do padrão.

A RTSJ acrescenta ao Java padrão as seguintes características (DIBBLE, 2002):

- adiciona threads de tempo real;
- permite a execução de código sem influência do coletor de lixo;
- implementa um gerenciador de eventos assíncronos e um mecanismo para transferência assíncrona de controle entre threads; e
- controla a localização de objetos e o acesso à memória em endereços físicos.

Existem diversas implementações da RTSJ, tanto nas esferas acadêmicas quanto comerciais. Alguns exemplos de desenvolvedores são Oracle, TimeSys e AICAS. A implementação utilizada neste trabalho foi o Java SE Real-time (Java RTS), desenvolvida pela SUN.

A seguir, são descritas algumas das principais características do RTSJ que são utilizadas neste trabalho.

2.3.1 Objeto Escalonável

A RTSJ introduz o conceito de objeto escalonável, que pode ser uma thread tempo real, ou uma thread tempo real não-*heap*, ou ainda um tratador de eventos assíncronos. Em sua construção, ela abstrai a unidade de escalonamento por trás do `java.lang.Thread`. Entidades que são escalonáveis são instâncias da interface `Schedulable`, que estende `Runnable` (Figura 2). Com isto, as aplicações podem imbuir os objetos escalonáveis com valores que caracterizam seu comportamento esperado e determinar seus requisitos temporais. Mais importante ainda, permite que as threads fiquem sob controle do escalonador.

Toda implementação da especificação deve incluir quatro definições de classes que implementam a interface `Schedulable`: `RealtimeThread (RTT)`, `NoHeapRealtimeThread (NHRT)`, `AsyncEventHandler (AEH)` e `BoundAsyncEventHandler (BAEH)`, cuja hierarquia está demonstrada na Figura 3.

2.3.2 Thread de Tempo Real

A criação de uma thread tempo real se dá pela instanciação de um objeto `RealtimeThread`, que possui associado:

- `ReleaseParameters`: parâmetros de liberação, tal como deadline;
- `Scheduler`: um escalonador da thread;

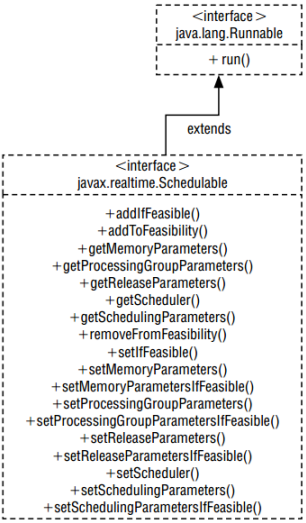

Figura 2: RTSJ - Interface de `Schedulable` (BRUNO; BOLLELLA, 2009)

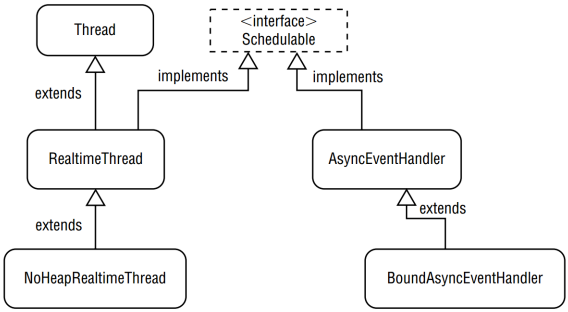

Figura 3: RTSJ - Objetos Escalonáveis (BRUNO; BOLLELLA, 2009)

- `SchedulingParameters`: parâmetros de escalonamento, tal como prioridade; e
- `MemoryParameters`: parâmetros de memória, tal como qual memória será utilizada.

Através do `ReleaseParameters` é possível especificar o comportamento de escalonamento da thread utilizando uma de suas subclasses: `PeriodicParameters` para uma tarefa periódica, `Aperiodic-`

`Parameters` para aperiódica, ou `SporadicParameters` para esporádica. Além dos parâmetros descritos, cada subclasse tem um tratador de perda de deadline (`DeadlineMissHandler`) e outro de sobrecarga (`CostOverrunHandler`).

O escalonador (`Scheduler`) padrão utilizado é baseado em prioridade. É utilizado através da classe `SchedulingParameters` e sua subclasse `PriorityParameters`. No Java RTS as prioridades variam entre 11 e 58, sendo que a mais baixa prioridade de uma thread tempo real ainda é superior às prioridades do coletor de lixo, do compilador JIT e de qualquer outra thread do Java convencional rodando no sistema (BRUNO; BOLLELLA, 2009), cujas prioridades variam entre 1 e 10.

O contexto de memória padrão é a *heap* e é definida pela classe `HeapMemory`, subclasse de `MemoryArea`. Além desta subclasse, também existem: `ScopedMemory`, `ImmortalMemory` e `ImmortalPhysicalMemory`.

A `NoHeapRealtimeThread` é uma especialização de `RealtimeThread` e, como seu nome indica, é uma thread que não pode ser alocada, nem referenciar variáveis, na memória *heap*. Ou seja, deve ser executada em um dos demais tipos de região de memória.

2.3.3 Tratador de Eventos Assíncronos

Eventos são acontecimentos que podem ocorrer tanto dentro quanto fora da JVM, tais como eventos de nível de sistema operacional, interrupções de hardware ou eventos definidos pela aplicação. Tratadores de eventos assíncronos são designados para tratar cada um destes eventos.

No RTSJ o tratador de eventos assíncronos é representado por duas classes: `AsyncEvent` e `AsyncEventHandler`. A primeira representa o evento propriamente dito e a segunda é um objeto escalonável executado quando o evento relacionado é disparado. Estes tratadores, assim como as threads, possuem parâmetros de liberação (`ReleaseParameters`) e parâmetros de escalonamento (`SchedulingParameters`).

Um tratador de eventos pode ser vinculado à uma thread através da classe `ReleaseParameters`. O código será executado de acordo com os parâmetros de tempo real especificados, sem a necessidade de criar explicitamente um tratador para cada evento, já que a JVM cuidará disto. A única exceção é o `BoundAsyncEventHandler`, que permite vincular um tratador de evento à uma thread específica.

3 TRABALHOS CORRELATOS

Desde antes mesmo de Heródoto, pai da história, o homem tem utilizado os registros de eventos passados para prever eventos futuros, tais como as estações climáticas, importantes para as plantações. E na Idade Contemporânea as informações passadas ainda são utilizadas para previsões. Em (MOORE, 1965), por exemplo, o autor analisa estatisticamente as informações de anos anteriores (aumento do custo e da quantidade de transistores em um circuito integrado) e define o que mais tarde ficou conhecido como Lei de Moore.

Diferentes tipos de previsões são usados em vários campos de conhecimento. Nos sistemas computacionais a literatura apresenta algoritmos para previsão de desempenho e tempo de resposta em diversas áreas de aplicação, tais como engenharia de software, sistemas autônomos, sistemas de banco de dados, redes e sistemas multimídia.

O artigo (ROSENBLATT, 1957) é um dos primeiros trabalhos que realiza uma previsão de desempenho, mencionando o uso de regressão linear. Anos mais tarde, em (BRYAN; SHEMER, 1969) os autores apresentam um modelo matemático que considera várias informações do sistema e calcula uma previsão de tempo de resposta. Em (BOYSE; WARN, 1975) um modelo analítico de previsão de desempenho de um sistema base é apresentado, além de demonstrar como ajustar o modelo para uso em outros sistemas. E em (HAMMER; CHAN, 1976) técnicas de suavização exponencial são usadas para calcular uma média estatística observada em diferentes períodos de tempo, a fim de prever características futuras para a utilização de um algoritmo heurístico que selecione automaticamente o índice de um sistema de gestão de banco de dados auto-adaptativo.

Os artigos (BROWN; ECKHOUSE JR.; ESTABROOK, 1977) e (GOEL; OKUMOTO, 1978) resolvem o mesmo problema de previsão de desempenho, mas agora aplicado a sistemas de tempo real. O trabalho de Devarakonda (1988) apresenta uma abordagem estatística para previsão de recursos através de um modelo de transição de estado para caracterizar o uso de recursos de cada programa em suas execuções passadas. Outros trabalhos ((GOSWAMI; IYER; DEVARAKONDA, 1989) (DEVARAKONDA; IYER, 1989) (DOWNEY, 1996)) utilizam esta tese como base, adequando e aperfeiçoando a abordagem, culminando em (SMITH; FOSTER; TAYLOR, 2004), que faz uma previsão de tempo de execução a partir de informações históricas e foi utilizado nas pesquisas de (TATIBANA; MONTEZ; OLIVEIRA, 2007) e (PLENTZ; MONTEZ; OLIVEIRA,

2008b), ambos descritos mais adiante neste capítulo.

Em (MAK; LUNDSTROM, 1990) é proposto um método para previsão de desempenho de uma classe de computação paralela que pode ser utilizada também em sistemas concorrentes. Este artigo foi utilizado por (SCHOPF, 1997), sendo este mencionado em (SCHOPF; BERMAN, 1998) e sequencialmente por (WOLSKI; SPRING; HAYES, 1998), (DINDA, 2001) e (LITKE; TSERPES; VARVARIGOU, 2005). Eles abordaram a previsão de carga do sistema, passando por sistemas distribuídos e grids computacionais e utilizando métodos de correlação, séries temporais e redes neurais. Esse último artigo, juntamente com (SMITH; FOSTER; TAYLOR, 2004) já citado anteriormente, foi utilizado por (PRODAN; NAE, 2009), sendo este último descrito mais adiante.

Alguns outros trabalhos que utilizam dados históricos para fazer uma previsão estão descritos abaixo.

O artigo (FENG; YINGYING; NIANBO, 2009) apresenta uma abordagem de previsão de curto prazo de tráfego veicular que utiliza dados históricos de tráfego, um modelo de previsão baseado em algoritmo *back propagation* para rede neural e compreensão de linguagem natural para representar eventos de trânsito. Além disto, é proposto um *framework* para combinar *Geographic Information System* (GIS), servidor de previsão de tráfego e sistema de gerenciamento de banco de dados para implementar a orientação dinâmica de rota. Foi realizada uma análise experimental em Pequim (República Popular da China), com mais de 10000 taxis equipados com GPS (*Global Positioning System*), viajando por mais de 2000 estradas principais e mais de 500 relatos de eventos de tráfego coletados na *Beijing Traffic Radio*. A abordagem provou ser uma boa solução para o serviço de informações públicas de viagem e mapas web dinâmicos.

Em (PRODAN; NAE, 2009) é proposto um método para o distribuição e escalonamento dinâmico de recursos para *Massively Multiplayer Online Games* (MMOGs) de tempo real em grids computacionais. Inicialmente, um serviço de previsão de carga antecipa a futura distribuição das entidades no mundo do jogo a partir de dados históricos utilizando um método baseado em rede neural. A partir disto, um modelo genérico de análise de carga de jogo é utilizado para prever futuros *hot-spots* que congestionam os servidores do jogo e tornam o ambiente fragmentado e impossível de jogar. Com base nas informações da previsão de carga, um serviço de distribuição de recursos executa a distribuição de carga dinâmica, o balanceamento e a migração de entidades que mantêm os servidores de jogos razoavelmente carregados de modo que os requisitos de *Quality of Service* (QoS) em tempo real são

mantidos.

O artigo (PERRONE et al., 2009) propõe uma metodologia para a estimativa de uma distribuição de probabilidade de tempo de execução, no contexto de sistemas de tempo real distribuídos baseados em componentes COTS (*Commercial off-the-shelf*), onde nenhum acesso ao código interno do componente é assumido (abordagem caixa-preta). Para realizar a estimativa, o método faz a análise do tempo de resposta e do tempo de comunicação entre componentes utilizando um componente *monitor* para a realização de medições. Essa metodologia foi avaliada através de um estudo de caso que demonstrou uma boa aproximação entre as distribuições de probabilidade estimada e medida.

A proposta descrita em (TATIBANA; MONTEZ; OLIVEIRA, 2007) determina a probabilidade de uma tarefa cumprir seu deadline em um sistema embutido que executa várias aplicações diferentes. O sistema mantém dois históricos, cada um com os tempos de resposta de cada serviço, juntamente com a informação sobre o estado do sistema (normal ou sobrecarregado) no momento da chegada da tarefa. Quando uma tarefa chega ao sistema e solicita um serviço, o tempo de resposta específico daquele serviço para a carga que o sistema se encontra no momento atual é usado para fins de previsão. A probabilidade de a tarefa alcançar seu deadline é calculada usando o registro histórico como uma função de massa de probabilidade. Uma das diferenças em relação aos mecanismos propostos nesta dissertação é que estes foram desenvolvidos e testados em uma linguagem de programação para sistemas de tempo real (Java RTS), além de utilizar regressão linear para calcular a relação entre os dados históricos.

O mecanismo de previsão de perda de deadline ASQ (*Aperiodic Server Queue Length*)(PLENTZ; MONTEZ; OLIVEIRA, 2008b) é um mecanismo para sistemas de tempo real implementados a partir de threads distribuídas. Ele relaciona informações conhecidas de ativações passadas com informações conhecidas em tempo de execução, através do uso de regressão linear. O mecanismo utiliza a composição da fila do Servidor de Aperiódicas (deadlines locais das demais threads ativas) dos nodos que compõem os possíveis itinerários que a thread distribuída pode executar e também uma estrutura com os dados históricos para realizar o cálculo da previsão, além de informações armazenadas no nodo onde a previsão está sendo realizada. Este mecanismo faz com que todas as threads distribuídas ativas no sistema colaborem na atualização das informações armazenadas nos nodos do sistema, necessárias à previsão.

Um dos mecanismos de previsão proposto nesta dissertação tam-

bém utiliza regressão linear para correlacionar informações do histórico de execuções com informações conhecidas somente em tempo de execução. A diferença se reflete em dois aspectos: em (PLENTZ; MONTEZ; OLIVEIRA, 2008b) o contexto de execução é um sistema distribuído constituído de threads distribuídas, enquanto este trabalho se refere a um sistema local, monoprocessado ou multiprocessado, que utiliza threads locais definidas pela RTSJ. O segundo aspecto se refere à forma de armazenamento e atualização dos dados. Em (PLENTZ; MONTEZ; OLIVEIRA, 2008b) isso é realizado pelas threads distribuídas, enquanto neste trabalho esta atualização ocorre em etapas as quais são realizadas pelas threads locais de tempo real definidas na RTSJ.

Outros trabalhos, como os apresentados em (PLENTZ; MONTEZ; OLIVEIRA, 2011a), (PLENTZ; MONTEZ; OLIVEIRA, 2011b) e (LORBIESKI; PLENTZ; FRIEDRICH, 2012) também propõem algoritmos de previsão de perda de deadline para sistemas distribuídos de tempo real. As principais diferenças em relação a este trabalho são: (PLENTZ; MONTEZ; OLIVEIRA, 2011a) e (PLENTZ; MONTEZ; OLIVEIRA, 2011b) consideram sistemas distribuídos de tempo real e não fazem uso de um histórico de informações para compor uma previsão de perda de deadline. E em (LORBIESKI; PLENTZ; FRIEDRICH, 2012) é apresentado uma implementação do mecanismo ASQ utilizando a RTSJ.

A Tabela 1 apresenta um comparativo entre os trabalhos apresentados acima. Cada linha indica um trabalho e suas respectivas características. Em Plentz, Montez e Oliveira (2008b), por exemplo, foi aplicado em um sistema distribuído, a técnica aplicada para o cálculo da previsão foi regressão linear, ele é um sistema de tempo real, utiliza histórico, foi validado utilizando uma simulação desenvolvida em Java e o overhead pela adição do mecanismo é médio, quando comparado com os demais.

Como mostra a Tabela 1, a maioria dos trabalhos são simulações, onde as características do sistema são controladas, e desenvolvidos em linguagem de programação tradicionais, não específicas para tempo real. Além disto, a maioria utiliza sistemas distribuídos e aqueles que não utilizam, não especificam se são monoprocessados ou multiprocessados.

No próximo capítulo serão detalhados os mecanismos propostos neste trabalho.

Tabela 1: Comparativo entre os Trabalhos Correlatos

Trabalho	**Tipo de Sistema**	**Técnica Aplicada no Mecanismo**	**Sistema de Tempo Real?**	**Utiliza Histórico?**	**Tipo de Validação**	**Linguagem de Programação**	**Overhead**
Feng, Yingying e Nianbo (2009)	Intelligent Transportation System	Rede Neural com *Backpropagation*	Não	Sim	Estudo de Caso	Não identificado	Alto
Prodan e Nae (2009)	MMOGs em Grids	Rede Neural	Sim	Sim	Simulação	C++	Alto
Perrone et al. (2009)	Sistema baseado em Componentes COTS	Distribuição de Probabilidade	Sim	Sim	Simulação	C++	Médio
Tatibana, Montez e Oliveira (2007)	Sistema Embutido	Função de Massa de Probabilidade	Sim	Sim	Simulação	Java	Médio
Plentz, Montez e Oliveira (2008b)	Sistema Distribuído	Regressão Linear	Sim	Sim	Simulação	Java	Médio
Plentz, Montez e Oliveira (2011a)	Sistema Distribuído	Cálculo Matemático	Sim	Não	Simulação	Java	Baixo
Plentz, Montez e Oliveira (2011b)	Sistema Distribuído	Cálculo Matemático	Sim	Não	Simulação	Java	Muito Baixo
Lorbieski, Plentz e Friedrich (2012)	Sistema Distribuído	Regressão Linear	Sim	Sim	Simulação	Java RTS	Médio

4 PROPOSTAS DE MECANISMOS DE PREVISÃO DE PERDA DE DEADLINE PARA TRATADORES DE EVENTOS

Por causa da natureza aperiódica dos eventos e seus tratadores, é importante definir um mecanismo de previsão de perda de deadline para eles. Com uma boa previsão é possível tomar ações corretivas para aprimorar o desempenho do sistema. Considerando-se um tratador de eventos (*Event Handler* - EH), é possível, por exemplo, alterar a forma como EH irá tratar seus eventos se o previsor indicar uma perda de deadline.

Nesta dissertação são apresentados dois mecanismos de previsão: o Mecanismo de Previsão de Perda de Deadline Baseado na Folga (MBF), que utiliza dados fornecidos pelo sistema, em tempo de execução, para realizar a previsão; e o Mecanismo de Previsão de Perda de Deadline Baseado no Histórico (MBH), que utiliza um histórico de dados de execuções passadas, além de dados atuais, para a realização do cálculo.

A Figura 4 apresenta uma visão geral do sistema utilizado na proposta. Um **Evento** gerado externa, ou internamente, no **Sistema Local** é atendido por um **Tratador**. Este **Tratador** está sendo escalonado junto com as demais tarefas do sistema e faz uso de um **Mecanismo de Previsão** para calcular a sua probabilidade de concluir a computação antes do seu deadline.

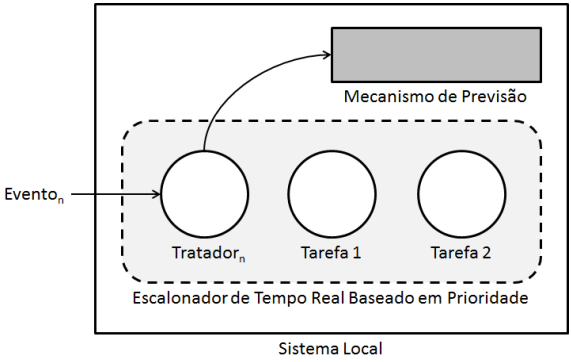

Figura 4: Visão Geral do Sistema

Nas próximas seções, para cada mecanismo proposto é apresentado o seu funcionamento, o algoritmo de utilização e uma análise do

overhead (Seções 4.1 e 4.2). Na Seção 4.3 são apresentadas as métricas usadas para avaliar os mecanismos.

4.1 MECANISMO DE PREVISÃO DE PERDA DE DEADLINE BASEADO NA FOLGA

O Mecanismo de Previsão de Perda de Deadline Baseado na Folga (MBF) mede a folga que a thread, que representa o tratador de eventos, possui para realizar sua computação e então realiza a previsão. Este mecanismo utiliza somente informações atuais da thread, sendo estas os tempos gastos pelas threads a espera pela execução. Este mecanismo é definido como:

$$P_k(MBF) = D_k - TEsp_k - TComp_k \tag{4.1}$$

$$Prob_k(MBF) = \begin{cases} 0 & \text{se } P_k(MBF) < 0 \\ P_k(MBF) & \text{se } 0 \leq P_k(MBF) \leq 1 \\ 1 & \text{se } P_k(MBF) > 1 \end{cases} \tag{4.2}$$

onde D_k é o deadline da thread k, $TEsp_k$ é o tempo de espera gasto pela thread k até o momento em que o mecanismo é acionado e $TComp_k$ é o tempo de computação da thread k.

Antes de iniciar a execução da thread, o mecanismo de previsão é acionado e o cálculo de previsão de perda de deadline é realizado e armazenado. Se necessário, ao final da execução do sistema, as métricas $E(MBF)$ e $PC(MBF)$ (descritas na Seção 4.3) avaliam a qualidade da previsão realizada.

Segue um exemplo de execução deste mecanismo. Um evento é disparado e uma thread m é criada para representar o tratador deste evento no tempo 100. O deadline deste tratador é de 60 milissegundos (ms) e seu tempo de computação é de 30ms. No tempo 125 a thread m assume o processador e o mecanismo é acionado:

$$\begin{gathered} P_m(MBF) = D_m - TEsp_m - TComp_m \\ P_m(MBF) = 60 - (125 - 100) - 30 \\ P_m(MBF) = 5 \\ P_m(MBF) > 1 \Rightarrow Prob_m(MBF) = 1 \end{gathered}$$

Assim, a probabilidade da thread m concluir a sua execução sem perder o deadline é 1. Se a thread assumisse o processador a partir

do tempo 130, por exemplo, a previsão indicaria probabilidade nula de conclusão sem perda de deadline.

4.1.1 Algoritmo de utilização

Considere o sistema composto por três componentes: **Sistema**, **Mecanismo** e **Métricas**. O primeiro representa o fluxo de execução normal da aplicação sem a interferência adicionada pelo cálculo da previsão e das métricas. O **Mecanismo** representa todas as adições necessárias para o cálculo da previsão, tal como o armazenamento de informações e o cálculo em si. Já o componente **Métricas** é responsável por armazenar algumas informações e avaliar a qualidade do mecanismo.

O algoritmo de utilização do MBF consiste de duas fases sequenciais: *Execução* e *Finalização*. A fase de *Execução* consiste no fluxo padrão do sistema, com a adição do levantamento de informações, cálculo da previsão e coleta de informações para as métricas. A Figura 5 apresenta o diagrama de sequência desta fase e a Tabela 2 descreve a função de cada mensagem trocada.

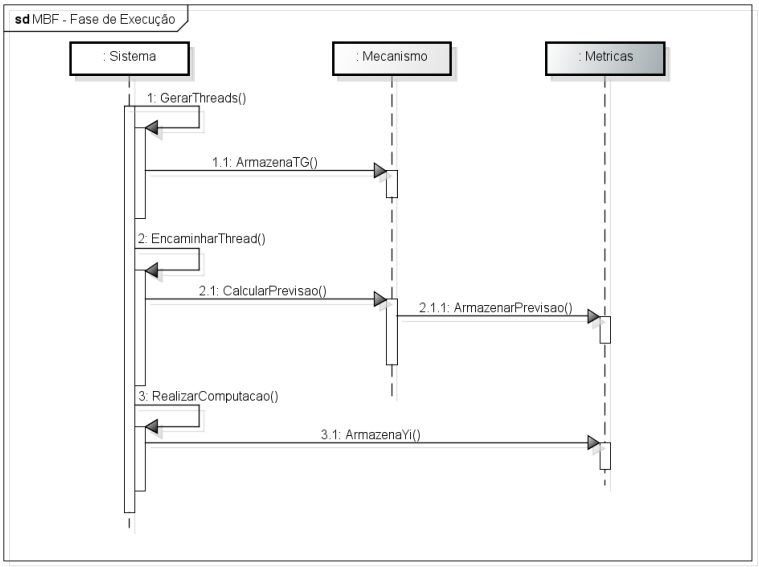

Figura 5: MBF - Algoritmo de Utilização - Fase de *Execução*

Tabela 2: MBF - Algoritmo de Utilização - Fase de *Execução*

Mensagem	Descrição
1: GerarThreads()	Assim que o sistema é iniciado, as threads que representam os tratadores de eventos começam a ser geradas.
1.1: ArmazenaTG()	É armazenado o instante de tempo em que cada thread é gerada.
2: EncaminharThread()	Cada thread é encaminhada para a fila de prontos do processador.
2.1: CalcularPrevisao()	Assim que uma thread assume o processador, o *MBF* é acionado para realizar os cálculos de previsão de perda de deadline.
2.1.1:ArmazenarPrevisao()	O resultado da previsão é armazenado.
3: RealizarComputacao()	Cada thread realiza a sua computação.
3.1: ArmazenaYi()	O tempo de resposta da thread é coletado e armazenado para ser utilizado com as métricas.

A fase de *Finalização* é responsável pela avaliação da qualidade das previsões realizadas. Para isto, cada métrica (descritas na Seção 4.3) acessa os resultados das previsões e realiza suas avaliações.

4.1.2 Overhead

A adição de um mecanismo de previsão de perda de deadline em um sistema deve ser previsto e estipulado na fase de projeto deste sistema. Assim, todo o overhead já será utilizado na definição das restrições temporais.

O MBF impõe duas fontes de overhead, decorrentes dos componentes adicionados:

- **Mecanismo:** requer que todo evento armazene o instante de tempo em que foi disparado. O deadline e o tempo de computação já são armazenados e disponibilizados pelos tratadores de eventos RTSJ. Além disto, requer também que sejam feitos os cálculos do mecanismo.

- **Métricas:** deve armazenar o resultado da previsão e o tempo de resposta da thread para serem utilizados pelas métricas de avaliação de qualidade. Se não existe a necessidade das avaliações, tanto o armazenamento destas informações, quanto a fase de *Finalização*, se fazem desnecessárias.

4.2 MECANISMO DE PREVISÃO DE PERDA DE DEADLINE BASEADO NO HISTÓRICO

O Mecanismo de Previsão de Perda de Deadline Baseado no Histórico (MBH) faz uso de dois tipos de informações: dados atuais da thread que representa o tratador de eventos, assim como o MBF, e um histórico de dados. O primeiro tipo de dado é conhecido pelo sistema somente em tempo de execução, a saber: tempo gasto pela thread a espera pela execução. O segundo apresenta informações sobre as execuções passadas da thread do tratador. Mais precisamente, é utilizado o tamanho da fila de threads deste tratador no estado de pronto do processador, ou seja, o número de threads que estão na frente da thread que representa este tratador de eventos.

Considerando as variáveis descritas anteriormente, este mecanismo é definido como:

$$P_k(MBH) = \frac{D_k - TEsp_k}{TRespE_k} \tag{4.3}$$

$$Prob_k(MBH) = \begin{cases} 0 & \text{se } P_k(MBH) < 0 \\ P_k(MBH) & \text{se } 0 \leq P_k(MBH) \leq 1 \\ 1 & \text{se } P_k(MBH) > 1 \end{cases} \tag{4.4}$$

onde D_k é o deadline da thread k, $TEsp_k$ é o tempo de espera gasto pela thread k até o momento em que o mecanismo é acionado e $TRespE_k$ é o tempo de resposta esperado da thread k até o final de sua execução.

Para gerar o tempo de resposta esperado ($TRespE_k$) são utilizadas informações de ativações passadas de todas as tarefas. O cálculo desta relação utiliza regressão linear, a qual é definida como (BARBETTA; REIS; BORNIA, 2004):

$$Y_i = \alpha + \beta X_i \tag{4.5}$$

onde Y_i representa o tempo de resposta estimado da thread k ($TRespE_k$) e X_i representa a quantidade de threads deste tratador que estão na fila de pronto do processador. As variáveis α e β representam estimativas para a relação entre os tempos de resposta e os respectivos tamanhos das filas de pronto do processador nas ativações passadas.

O método mais usual para o cálculo desta relação é com mínimos quadrados (BARBETTA; REIS; BORNIA, 2004), calculado a partir de um conjunto de observações (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), onde x_i repre-

senta o tamanho da fila de pronto do processador antes do início da execução da thread e y_i representa o tempo de resposta desta thread. O método dos mínimos quadrados é definido da seguinte forma:

$$\beta = \frac{n\sum(x_i y_i) - (\sum x_i)(\sum y_i)}{n\sum x_i^2 - (\sum x_i)^2} \tag{4.6}$$

$$\text{Se } (\beta < 0) \rightarrow \beta = 0$$

$$\alpha = \frac{\sum y_i - \beta \sum x_i}{n} \tag{4.7}$$

Antes de iniciar a execução da thread, o mecanismo de previsão é acionado e os cálculos de previsão de perda de deadline são realizados e armazenados. Se necessário, ao final da execução do sistema, as métricas $E(MBH)$ e $PC(MBH)$ (descritas na Seção 4.3) avaliam a qualidade da previsão realizada.

Segue um exemplo de execução deste mecanismo. Um evento é disparado e uma thread n é criada para representar o tratador deste evento no tempo 100. O deadline deste tratador é de 60ms e seu tempo de computação é de 30ms. No tempo 125 a thread n assume o processador e o mecanismo é acionado. Considerando que, segundo a regressão linear, o tempo de resposta esperado para esta thread ($TRespE_n$) é de 35ms, o cálculo da previsão segue abaixo:

$$P_n(MBH) = \frac{D_n - TEsp_n}{TRespE_n}$$
$$P_n(MBH) = \frac{60-(125-100)}{35}$$
$$P_n(MBH) = 1$$
$$0 \leq P_n(MBH) \leq 1 \Rightarrow Prob_n(MBH) = P_n(MBH) = 1$$

Assim, a probabilidade da thread n concluir a sua execução sem perder o deadline é 1. Se a thread assumisse o processador no tempo 130, por exemplo, a previsão indicaria $Prob_n(MBH) = 0,857$.

4.2.1 Algoritmo de utilização

Assim como no mecanismo anterior, o MBH utiliza os componentes **Sistema**, **Mecanismo** e **Métricas**. Entretanto, o **Mecanismo** transfere a responsabilidade de armazenar as informações para um novo componente, denominado **Histórico**. As demais responsabilidades se mantêm.

O algoritmo de utilização do MBH consiste de três fases sequenciais: *Inicialização*, *Execução* e *Finalização*. A fase de *Inicialização* consiste no fluxo padrão do sistema e tem a função de coletar os dados para gerar um histórico para a utilização no cálculo do mecanismo. A Figura 6 apresenta o diagrama de sequência desta fase e a Tabela 3 descreve a função de cada mensagem trocada.

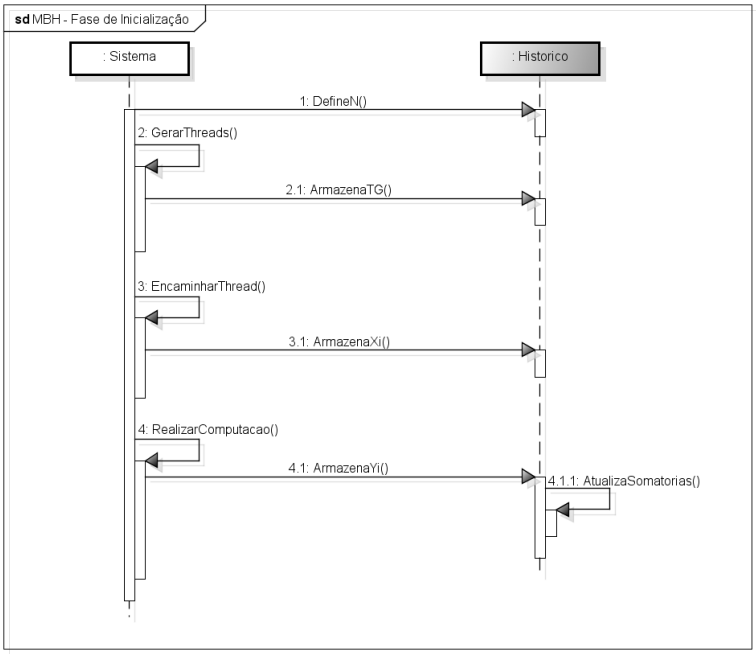

Figura 6: MBH - Algoritmo de Utilização - Fase de *Inicialização*

Quando n eventos são tratados, inicia-se a fase de *Execução*. Nesta fase o fluxo padrão do sistema é executado, com a adição do cálculo da previsão e coleta de informações para as métricas. A Figura 7 apresenta o diagrama de sequência desta fase e a Tabela 4 descreve a função de cada mensagem trocada.

A fase de *Finalização* é responsável pela avaliação da qualidade das previsões realizadas. Para isto, cada métrica (descritas na Seção 4.3) acessa os dados armazenados no **Histórico** e realiza suas avaliações.

A forma simplificada de execução do algoritmo, como apresentado acima, é sequencial. Entretanto, a fase de *Execução* pode ser rea-

Tabela 3: MBH - Algoritmo de Utilização - Fase de *Inicialização*

Mensagem	**Descrição**
1: DefineN()	Define-se a quantidade de eventos (n) que serão tratados em cada fase.
2: GerarThreads()	Assim que o sistema é iniciado, as threads que representam os tratadores de eventos começam a ser geradas.
2.1: ArmazenaTG()	É armazenado no **Histórico** o instante de tempo em que cada thread é gerada.
3: EncaminharThread	Cada thread é encaminhada para a fila de prontos do processador.
3.1: ArmazenaXi	O número de threads do tratador no estado de pronto do processador (x_i) é coletado e armazenado no **Histórico**.
4: RealizarComputacao()	Cada thread realiza a sua computação.
4.1 ArmazenaYi()	O tempo de resposta de cada thread (y_i) é coletado e armazenado no **Histórico** para ser utilizado pelo mecanismo.
4.1.1 AtualizaSomatorias()	Os valores de $\sum x_i$, $\sum y_i$, $\sum x_i^2$ e $\sum(x_i y_i)$ são atualizados.

Tabela 4: MBH - Algoritmo de Utilização - Fase de *Execução*

Mensagem	**Descrição**
1: CalculaAlfaBeta()	O *MBH* consulta o **Histórico** e faz o cálculo da relação entre os tempos de resposta e os respectivos tamanhos da fila de pronto do processador nas ativações passadas (α e β).
2: GerarThreads()	As threads que representam os tratadores de eventos começam a ser geradas.
3: EncaminharThread()	Cada thread é encaminhada para a fila de prontos do processador.
3.1: CalcularPrevisao()	Assim que uma thread assume o processador, o *MBH* é acionado para realizar os cálculos de previsão de perda de deadline.
3.1.1: ArmazenarPrevisao()	O resultado da previsão é armazenado no **Histórico**.
4: RealizarComputacao()	Cada thread realiza a sua computação.
4.1: ArmazenaYi()	O tempo de resposta de cada thread (y_i) é coletado e armazenado no **Histórico** para ser utilizado com as métricas.

lizada de forma iterativa, assim as variáveis de controle do mecanismo serão constantemente ajustadas. Para isto, a cada n eventos tratados, esta fase executa o fluxo apresentado no diagrama de sequência da Figura 8. A Tabela 5 descreve a função de cada mensagem trocada.

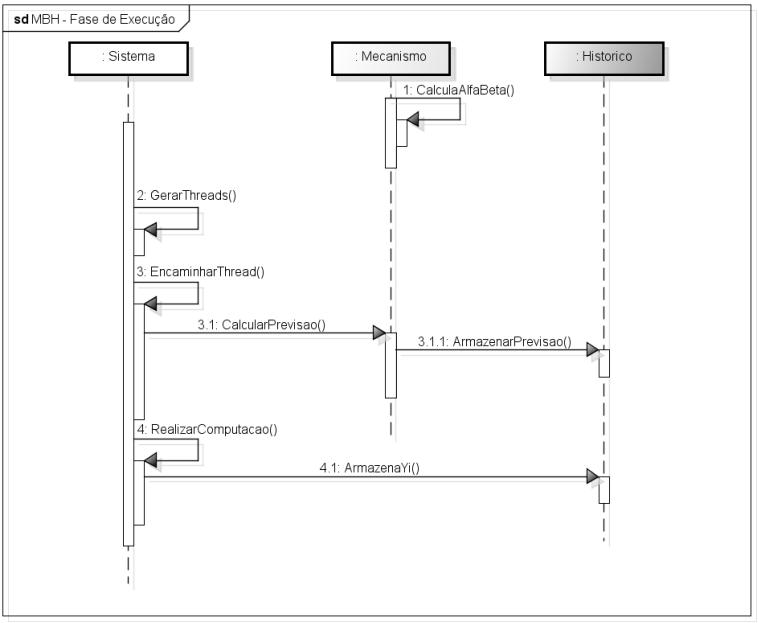

Figura 7: MBH - Algoritmo de Utilização - Fase de *Execução*

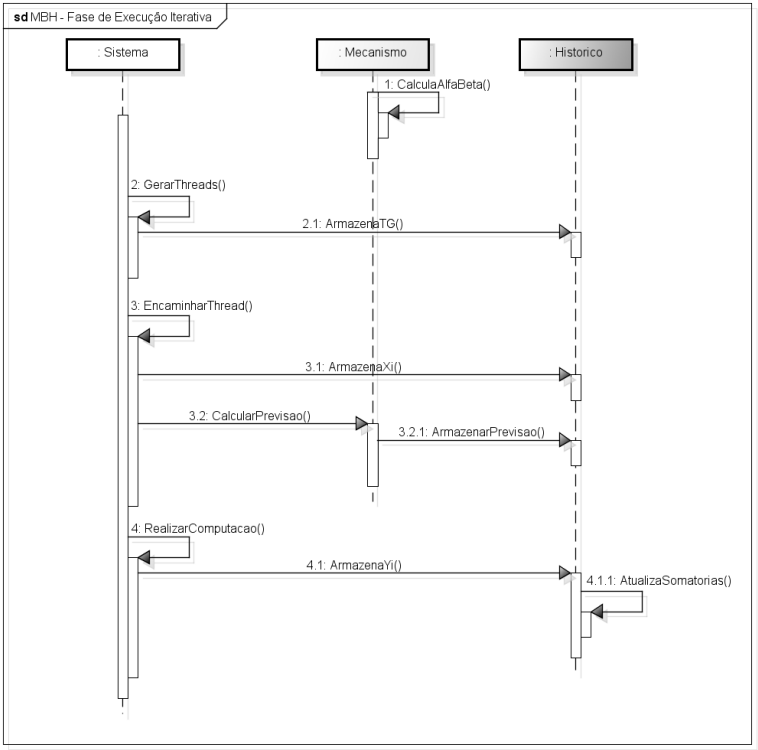

Figura 8: MBH - Algoritmo de Utilização - Fase de *Execução* Iterativa

Tabela 5: MBH - Algoritmo de Utilização - Fase de *Execução* Iterativa

Componente	**Descrição**
1: CalculaAlfaBeta()	O *MBH* consulta o **Histórico** e faz o cálculo da relação entre os tempos de resposta e os respectivos tamanhos da fila de pronto do processador nas ativações passadas (α e β).
2: GerarThreads()	As threads que representam os tratadores de eventos começam a ser geradas.
2.1: ArmazenaTG()	É armazenado no **Histórico** o instante de tempo em que cada thread é gerada.
3: EncaminharThread()	Cada thread é encaminhada para a fila de prontos do processador.
3.1: ArmazenaXi()	O número de threads do tratador no estado de pronto do processador (x_i) é coletado para o **Histórico**.
3.2: CalcularPrevisao()	Assim que uma thread assume o processador, o *MBH* é acionado para realizar os cálculos de previsão de perda de deadline.
3.2.1: ArmazenarPrevisao()	O resultado da previsão é armazenado no **Histórico**.
4: RealizarComputacao()	A thread realiza a sua computação.
4.1: ArmazenaYi()	O tempo de resposta de cada thread (y_i) é coletado e armazenado no **Histórico** para ser utilizado tanto com as métricas quanto com o mecanismo.
4.1.1: AtualizaSomatorias()	Os valores de $\sum x_i$, $\sum y_i$, $\sum x_i^2$ e $\sum(x_i y_i)$ são atualizados.

4.2.2 Overhead

Este mecanismo impõe as seguintes adições ao sistema:

- A quantidade de eventos (n) que serão tratados na fase de *Inicialização* e consequentemente em cada iteração da fase de *Execução*.
- O **Mecanismo**, responsável:
  - Pelos valores de: $\sum x_i$, $\sum y_i$, $\sum x_i^2$ e $\sum(x_i y_i)$, utilizados no cálculo da regressão linear e atualizados a cada evento tratado.
  - Pelos valores de α e β, calculados a cada início da fase de *Execução*.
  - Pelo cálculo do mecanismo em si.
- Um **Histórico** que relacione para cada evento disparado:
  - O instante de tempo em que a thread é gerada.
  - O tempo de resposta da thread (y_i).
  - O resultado da previsão.
- As **Métricas**, responsáveis:
  - Pelos cálculos das métricas de avaliação de qualidade.

O armazenamento do número de threads do tratador no estado de pronto do processador (x_i) não é necessário, pois é utilizado somente pelo **Mecanismo** na atualização das somatórias. O **Histórico** também não precisa armazenar o deadline e o tempo de computação, pois estas informações são disponibilizadas pelos tratadores de eventos RTSJ.

As **Métricas** utilizam os valores do tempo de resposta (y_i) e do resultado da previsão, ambos armazenados no **Histórico**, além do deadline da thread que representa o tratador de eventos. Se não existe a necessidade das avaliações de qualidade, não será necessário tanto o armazenamento destas informações, quanto a fase de *Finalização*.

4.3 MÉTRICAS DE QUALIDADE DAS PREVISÕES

Abaixo são descritas as métricas utilizadas para avaliar a qualidade das previsões realizadas pelos mecanismos propostos. Estas métricas são as mesmas definidas em (PLENTZ, 2008).

4.3.1 Métrica Taxa Relativa de Erro

Sendo z um mecanismo de previsão de perda de deadline, a Métrica Taxa Relativa de Erro ($E(z)$) calcula o erro associado com a previsão realizada por z em função do tempo de resposta da thread. Ela é definida como:

$$E_k(z) = \begin{cases} 1 - Prob_k(z) & \text{se } R_k \leq D_k \\ Prob_k(z) & \text{se } R_k > D_k \end{cases}$$

$$E(z) = \sum_{k=1}^{n_k} \frac{E_k(z)}{n_k} \tag{4.8}$$

onde $E_k(z)$ é o erro associado com a previsão realizada pelo mecanismo z para uma thread k, $Prob_k(z)$ representa a probabilidade definida pelo mecanismo z da thread k cumprir seu deadline, R_k e D_k são o tempo de resposta e o deadline da thread, respectivamente, e n_k é o número total de threads no sistema.

Os exemplos a seguir ilustram o comportamento desta métrica em três execuções do mecanismo de previsão z:

- z estima que a thread k irá perder seu deadline ($Prob_k(z) = 0$) o que realmente ocorre ($R_k > D_k$); então $E_k(z) = Prob_k(z) = 0$.

- z estima que existe uma chance de 84% da thread cumprir seu deadline ($Prob_l(z) = 0.84$), o que de fato ocorre ($R_l \leq D_l$); então $E_l(z) = 1 - 0.84 = 0.16$.

- z estima que existe uma chance de 50% da thread cumprir seu deadline ($Prob_m(z) = 0.50$); neste caso $E_m(z) = 0.50$ independentemente da thread cumprir ou não seu deadline.

- Por fim, o resultado da métrica é $E(z) = (E_k(z) + E_l(z) + E_m(z)/3) = (0 + 0.16 + 0.50)/3 = 0.22$.

É importante observar que esta métrica representa a convicção de um algoritmo de previsão sobre a capacidade da thread cumprir ou não um deadline (PLENTZ; MONTEZ; OLIVEIRA, 2008a). Um mecanismo perfeito de previsão de perda de deadline deve gerar um erro igual a zero ao longo de suas execuções.

4.3.2 Métrica Taxa de Previsões Corretas

A Métrica Taxa de Previsões Corretas ($PC(z)$) calcula o número de previsões corretas sobre o número total de previsões realizadas pelo mecanismo. Se o resultado da previsão realizada pelo mecanismo é igual ou maior que 50% e a thread não perdeu o deadline, então a métrica considera a previsão correta. O mesmo ocorre quando o resultado da previsão é inferior a 50% e a thread perde o deadline. Esta métrica é definida como:

$$\begin{array}{l} \text{Se } (Prob_k(z) < 50\%) \text{ e } (R_k > D_k) \\ \quad \text{Então } PrevisoesCorretas(z) += 1 \\ \text{Se } (Prob_k(z) \geq 50\%) \text{ e } (R_k \leq D_k) \\ \quad \text{Então } PrevisoesCorretas(z) += 1 \end{array}$$

$$PC(z) = PrevisoesCorretas(z)/n \tag{4.9}$$

onde $Prob_k(z)$ é a probabilidade definida pelo mecanismo z da thread k cumprir seu deadline, R_k e D_k são o tempo de resposta e o deadline da thread, respectivamente, $PrevisoesCorretas(z)$ é o número de previsões corretas do mecanismo e n é o número total de previsões, sendo uma para cada thread no sistema.

Mantendo as execuções e valores do exemplo da seção anterior (Seção 4.3.1), a métrica se comporta da seguinte forma:

- z estima que a thread k irá perder seu deadline ($Prob_k(z) = 0$) o que realmente ocorre ($R_k > D_k$); então $PrevisoesCorretas(z) = 1$.

- z estima que existe uma chance de 84% da thread cumprir seu deadline ($Prob_l(z) = 0.84$), o que de fato ocorre ($R_l \leq D_l$); então $PrevisoesCorretas(z) = 2$.

- z estima que existe uma chance de 50% da thread cumprir seu deadline ($Prob_m(z) = 0.50$), mas a thread não cumpre seu deadline ($R_k > D_k$); então $PrevisoesCorretas(z) = 2$.

- Por fim, o resultado da métrica é $PC(z) = 2/3 = 0.67$.

Quanto mais próximo de 1 for o resultado desta métrica, melhor será a previsão realizada pelo mecanismo.

4.4 CONCLUSÕES

Neste capítulo dois mecanismos de previsão de perda de deadline foram descritos: MBF e MBH. O primeiro utiliza dados da própria thread (deadline e tempo de computação) e um dado conhecido somente em tempo de execução (tempo de espera na fila de prontos do tratador) para o cálculo da previsão e acarreta dois componentes fontes de overhead (Mecanismo e Métricas), mas que podem ser reduzidas para somente duas funções (cálculo da previsão e o armazenamento do tempo de espera).

O MBH faz uso de dados das execuções passadas da thread e de dados conhecidos somente em tempo de execução para o cálculo da previsão. Para isto, é utilizado regressão linear e mínimos quadrados para gerar um tempo de resposta estimado. Este mecanismo acarreta três componentes que são fontes de overhead, mas que podem ser reduzidos para dois, se necessário.

Para avaliar a qualidade das previsões destes mecanismos, na fase de *Finalização* são utilizados duas métricas ($E(z)$ e $PC(z)$). A métrica $E(z)$ verifica o quão distante de uma previsão exata foi a previsão realizada. Se a thread cumpriu o deadline o erro será o que faltou para o previsor alcançar uma previsão 100% correta, senão, o erro será o próprio valor da previsão. A métrica $PC(z)$ contabiliza o número de previsões corretas de acordo com um critério especificado. Este critério define que se o valor resultante do cálculo da previsão for maior que 50%, assume-se que a thread irá cumprir seu deadline, senão, assume-se que ela não irá cumprir seu deadline. Ao final, a métrica contabiliza a quantidade de acertos.

No próximo capítulo são descritos os testes realizados e analisados os resultados encontrados com estes dois mecanismos propostos.

5 AMBIENTES, TESTES E RESULTADOS

Testes foram realizados a fim de avaliar a qualidade das previsões feitas com os mecanismos propostos. Nas próximas seções são descritos um estudo de caso, o ambiente de testes, o fluxo de execução, características da implementação, características das cargas do sistema e os resultados dos testes.

5.1 ESTUDO DE CASO - CRUISE CONTROL

Em Wellings (2005) é apresentado um estudo de caso do *Cruise Control* (CC), conhecido popularmente no Brasil como Piloto Automático. Este é um sistema de tempo real de apoio à condução de veículos automotores que permite ao motorista definir e manter a velocidade do veículo sem que seja necessária a sua intervenção, reduzindo assim o desgaste do automóvel, aumentando a economia de combustível e reduzindo também a fadiga do motorista em viagens de longa duração.

Os comandos do sistema podem ser enviados de forma explícita ou implícita. Os comandos explícitos são dados pela interface do condutor, normalmente uma alavanca ao lado do volante, e os comandos implícitos são emitidos quando o condutor troca de marcha ou pressiona o freio. Em ambos os casos o sistema é desativado.

A velocidade do carro é medida pelo CC através da rotação do eixo que aciona as rodas traseiras. Este eixo gera uma interrupção para cada rotação e o sistema tem uma configuração padrão para o número de interrupções que devem ser gerados para cada quilômetro percorrido.

Se o sistema falha, por exemplo, em verificar a velocidade a tempo para o algoritmo de controle, o último valor armazenado deverá ser utilizado. O CC ainda pode funcionar de forma aceitável, tendo em vista que a variação da velocidade entre a última verificação e a atual é pequena, entretanto, sua eficiência é degradada. O resultado da verificação atrasada é inútil para o sistema e pode ser descartado, característica presente em sistemas de tempo real firme. Outra característica que levou à este estudo de caso é que o CC pode ser implementado utilizando tanto uma, quanto várias, *Engine Control Units* (ECU), caracterizando um ambiente monoprocessado ou multiprocessado.

Todos os dispositivos são mapeadas em memória e têm associado registradores de controle e de dados. Sensores no automóvel detectam alterações de estado e geram interrupções necessárias para o sistema de

controle.

O sistema apresentado em Wellings (2005) é constituído pelos seguintes componentes principais:

- tratadores de interrupção para as interrupções causadas pela alavanca de comando (*lever*), freio (*brake*), motor (*engine*), eixo da roda (*shaft*) e transmissão (*gear*);
- uma thread periódica para controlar o acelerador;
- uma thread periódica para monitorar a velocidade;
- um controlador para coordenar o sistema.

A interação entre esses componentes é representada pelo diagrama de colaboração apresentado na Figura 9.

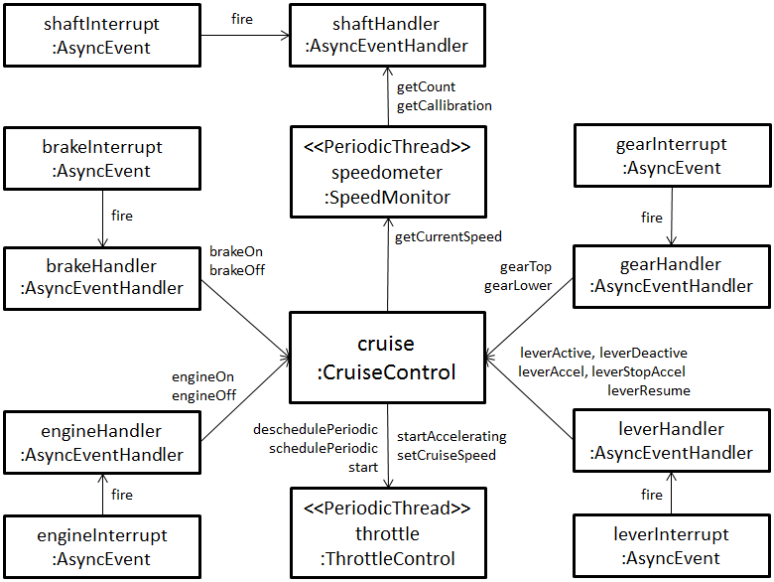

Figura 9: Cruise Control (WELLINGS, 2005)

5.2 AMBIENTES DOS TESTES

O sistema utilizado para os testes foi projetado a partir da generalização do CC, buscando abranger uma maior gama de cenários.

Foi desenvolvido em Java RTS versão 2.2 e executado em um computador de propósito geral, onde outros processos de controle do sistema operacional também executam, com um Ubuntu 10.4 kernel RT versão 2.6.31-11. Foram realizados testes em um ambiente monoprocessado e em um multiprocessado com quatro processadores (*cores*). Este sistema é composto por:

- *Gerador* de *Sinais*: tarefa periódica pertencente à classe `RealtimeThread`. Tem a função de gerar *Sinais*, simulando, por exemplo: um sinal POSIX, uma E/S em arquivo ou mesmo uma interrupção causada por um dos componentes do CC. Estes *Sinais* disparam aleatoriamente *Eventos*, impondo a característica aperiódica aos mesmos.
- *Evento*: pertence a classe `AsyncEvent`. Representa o evento disparado e cada instância possui um *Tratador* vinculado.
- *Tratador* de *Eventos*: tarefa aperiódica que pertence à classe `BoundAsyncEventHandler`. Tem a função de tratar o evento disparado e cada instância é responsável por somente um *Evento*.

Para poder abranger uma ampla gama de combinações de ambientes para os testes, algumas decisões feitas pelo sistema são tomadas de forma aleatória e algumas variáveis numéricas (especificadas na Tabela 6) também são aleatórias (seguem uma distribuição uniforme). O número de *Sinais* gerados, por exemplo, variam de 10 a 150 por *Geradores*, sendo que o números de *Geradores*, *Eventos* e seus respectivos *Tratadores*, variam de 5 a 20 e o tempo de computação dos *Tratadores* variam de 10 a 150 milissegundos (ms).

Tabela 6: Testes - Variáveis Aleatórias

Variável Aleatória	**Valor Mín.**	**Valor Máx.**
Tempo de Computação do Tratador (tcT)	10 ms	150 ms
Deadline do Tratador (dT)	$2.tcT$ ms	$10.tcT$ ms
Prioridade do Tratador (rT)	11	58
Período do Gerador (pG)	tcT ms	$50.tcT$ ms
Quantidade de Geradores (qtG)	5	20
Quantidade de Eventos/Tratadores (qtE)	5	20
Quantidade de Sinais (qtS)	10	150

As prioridades dos *Tratadores* (rT) variam entre o valor mínimo e máximo definido pelo Java RTS. O deadline dos *Tratadores* (dT) são divididos em três tipos e nas seguintes proporções:

1. Folgado (20%): o deadline será cumprido.
2. Justo (20%): possui deadline apertado;
3. Suficiente (60%): possui folga suficiente para executar se o sistema está estável;

Assim, existe uma grande chance de 20% dos eventos conseguirem cumprir seu deadline, independente da carga do sistema, 20% não conseguirão cumprir na maioria dos casos e 60% irão cumprir se o sistema estiver estável.

A Tabela 7 apresenta os valores utilizados para os testes específicos, descritos nas próximas seções. Considere T1, T2, T3, T4 e T5 os cinco *Tratadores* utilizados.

Tabela 7: Testes - Variáveis Fixas

Característica	**T1**	**T2**	**T3**	**T4**	**T5**
Tempo de Computação do Tratador (tcT)	50ms	150ms	250ms	350ms	450ms
Deadline do Tratador (dT)	400ms	300ms	1000ms	1400ms	1800ms
Prioridade do Tratador (rT)	55	45	35	25	15
Período do Gerador (pG)	500ms				
Quantidade de Geradores (qtG)	10				
Quantidade de Eventos/Tratadores (qtE)	5				
Quantidade de Sinais (qtS)	100				

Além destas características, o sistema ainda possui tarefas periódicas de tempo real (pertencentes à classe `RealtimeThread`) e tarefas do Java convencional (pertencentes à classe `Thread`), representando outras funções realizadas pelo sistema e pelo ambiente, inclusive as threads periódicas para controlar o acelerador e monitorar a velocidade no CC. A quantidade destas tarefas é a metade da quantidade de *Eventos* (qtE), os deadlines variam na mesma proporção dos deadlines dos *Tratadores* (dT) e as prioridades são as definidas como normais (26 para as tarefas periódicas de tempo real e 5 para as tarefas do Java convencional).

Outra característica importante é o algoritmo de escalonamento. O algoritmo padrão utilizado pela Java RTS é preemptivo e baseado em prioridade. Desta forma, foi definido que os *Geradores* possuem prioridade máxima de execução, enquanto que os *Tratadores* possuem uma prioridade aleatória. Sendo assim, os *Geradores* ficam enviando

Sinais de forma periódica sem que um *Tratador* tome o seu lugar no processador na maioria do tempo. Isto só não ocorrerá se o *Tratador* conseguir aleatoriamente a prioridade máxima.

5.3 FLUXOS DE EXECUÇÃO

O diagrama de sequência do fluxo padrão de execução do sistema de teste é apresentado na Figura 10 e descrito na Tabela 8.

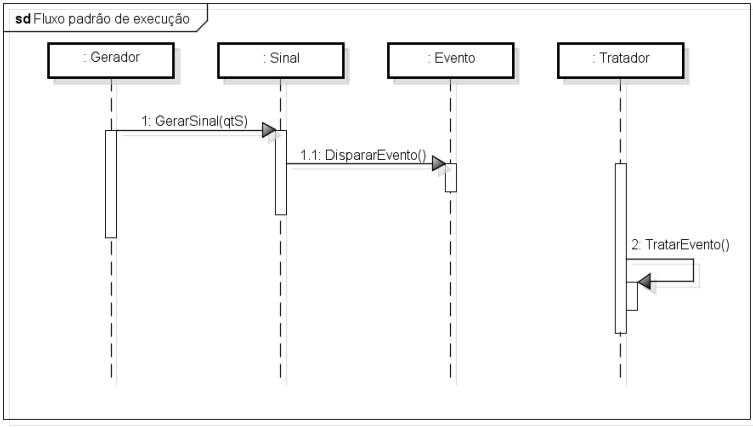

Figura 10: Fluxo Padrão de Execução do Sistema

Tabela 8: Fluxo Padrão de Execução do Sistema

Mensagem	**Descrição**
1: GerarSinal(qtS)	Assim que os sistema é iniciado, cada um dos *qtG Geradores* (G) iniciam a geração de *qtS Sinais* (S).
1.1: DispararEvento()	Cada *Sinal* dispara aleatoriamente um *Evento* (E).
2: TratarEvento()	Cada *Tratador* (T) realiza o tratamento de seu respectivo *Evento*.

5.3.1 Fluxo de execução com o MBF

O algoritmo é aplicado no sistema de teste e segue as duas fases do MBF. Na fase de *Execução* o fluxo padrão do sistema é realizado com

a adição do cálculo da previsão de perda de deadline, como ilustrado no diagrama de sequência da Figura 11 e descrito de forma simplificada na Tabela 9.

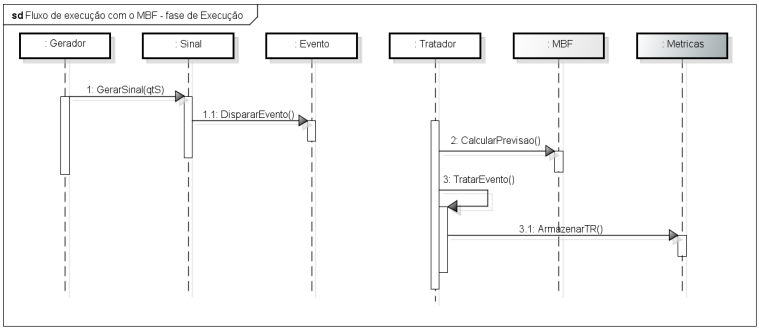

Figura 11: Fluxo de Execução com o MBF - Fase de *Execução*

Tabela 9: Fluxo de Execução com o MBF - Fase de *Execução*

Componente	**Descrição**
1: GerarSinal(qtS)	Assim que os sistema é iniciado, cada um dos *qtG Geradores* iniciam a geração de *qtS Sinais*.
1.1: DisparарEvento()	Cada *Sinal* dispara aleatoriamente um *Evento*.
2: CalcularPrevisao()	O *Mecanismo* consulta o instante de tempo em que a thread foi gerada e realiza os cálculos de previsão de perda de deadline, sendo o resultado armazenado.
3: TratarEvento()	Cada *Tratador* realiza o tratamento de seu respectivo *Evento*.
3.1: ArmazenarTR()	O tempo de resposta do *Tratador* é coletado e armazenado para ser utilizado pelas *Métricas* (M).

Durante a *Finalização*, as *Métricas* $E(MBF)$ e $PC(MBF)$ são calculadas a partir dos dados armazenados.

5.3.2 Fluxo de execução com o MBH

O algoritmo é aplicado no sistema de teste e segue as três fases do MBH sem iteração. Na primeira são coletados os dados para gerar o *Histórico*, conforme ilustrado pela Figura 12 e descrito de forma simplificada na Tabela 10.

Na fase de *Execução* um ciclo similar é realizado. Adiciona-se

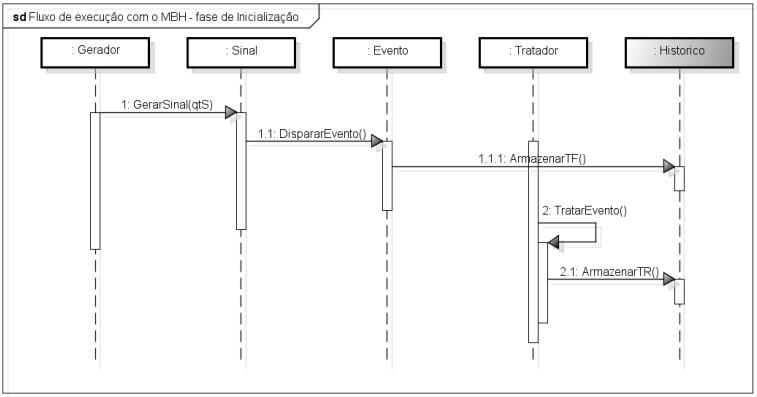

Figura 12: Fluxo de Execução com o MBH - Fase de *Inicialização*

Tabela 10: Fluxo de Execução com o MBH - Fase de *Inicialização*

Componente	**Descrição**
1: GerarSinal(qtS)	Assim que os sistema é iniciado, cada um dos *qtG Geradores* iniciam a geração de *qtS Sinais*.
1.1: DispararEvento()	Cada *Sinal* dispara aleatoriamente um *Evento*.
1.1.1: ArmazenarTF()	O instante de tempo em que o *Evento* foi disparado e número de threads no estado de pronto do processador são coletados para o *Histórico* (H).
2: TratarEvento()	Cada *Tratador* realiza o tratamento de seu respectivo *Evento*.
2.1: ArmazenarTR()	O tempo de resposta é coletado e os valores dos somatórios são atualizados.

a ele o cálculo da previsão de perda de deadline, como ilustrado pela Figura 13 e descrito de forma simplificada na Tabela 11.

Durante a *Finalização*, as *Métricas* $E(MBH)$ e $PC(MBH)$ são calculadas a partir dos dados armazenados no *Histórico*.

5.4 CARACTERÍSTICAS DA IMPLEMENTAÇÃO

Os mecanismos MBF e MBH foram implementados como um módulo de software que recebe as informações necessárias e retorna o resultado da previsão. Este módulo possui três classes principais:

- **Mecanismo**: responsável pelo cálculo da regressão linear, utili-

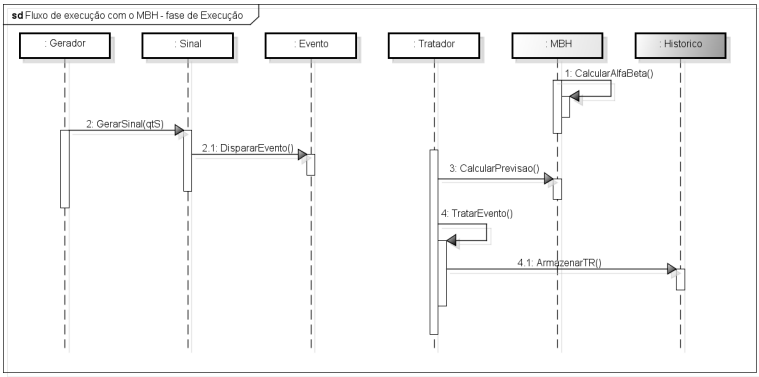

Figura 13: Fluxo de Execução com o MBH - Fase de *Execução*

Tabela 11: Fluxo de Execução com o MBH - Fase de *Execução*

Componente	**Descrição**
1: CalcularAlfaBeta()	O *Mecanismo* consulta o *Histórico* e faz o cálculo de α e β.
2: GerarSinal(qtS)	Assim que os sistema é iniciado, cada um dos *qtG Geradores* iniciam a geração de *qtS Sinais*.
2.1: DispararEvento()	Cada *Sinal* dispara aleatoriamente um *Evento*.
3: CalcularPrevisao()	O *Mecanismo* é acionado para realizar os cálculos de previsão de perda de deadline e o resultado é armazenado no *Histórico*.
4: TratarEvento()	Cada *Tratador* realiza o tratamento de seu respectivo *Evento*.
4.1: ArmazenarTR()	O tempo de resposta do *Tratador* é coletado e armazenado no *Histórico*.

zado somente pelo MBH, e pelo mecanismo de previsão de perda de deadline.

- **Métricas**: responsável pelo cálculo da de $E(z)$ e $PC(z)$.
- **Histórico**: responsável pelo armazenamento de informações das execuções passada, utilizado somente pelo MBH, e pelas informações necessárias para o cálculo das *Métricas*.

Uma das informações necessárias para realizar a previsão do MBH é o tamanho da fila de prontos do processador. Muitos sistemas não possuem acesso a esta informação, entretanto, a RTSJ especificou uma solução para prover esta informação aos desenvolvedores de siste-

mas de tempo real. A especificação define a implementação da função `getPendingFireCount()`, que deve retornar a quantidade de eventos pendentes associados a um `AsyncEventHandler`, neste caso, aos *Tratadores*. Assim, sempre que um evento é disparado a classe controladora do sistema executa esta função em todos os *Tratadores* e armazena a soma dos resultados (a quantidade total de eventos na fila de prontos do processador) no único *Histórico* do sistema.

Um exemplo de execução com esta implementação é apresentado na Figura 14. A Tabela 12 mostra os resultados do teste, considerando as variáveis definidas na Tabela 7, pág. 66. Cada ponto representa um *Sinal* gerado e a relação entre o tamanho da fila de prontos do processador e o tempo total desde o disparo do evento até a finalização do tratamento. A linha diagonal apresenta a regressão linear calculada no respectivo teste.

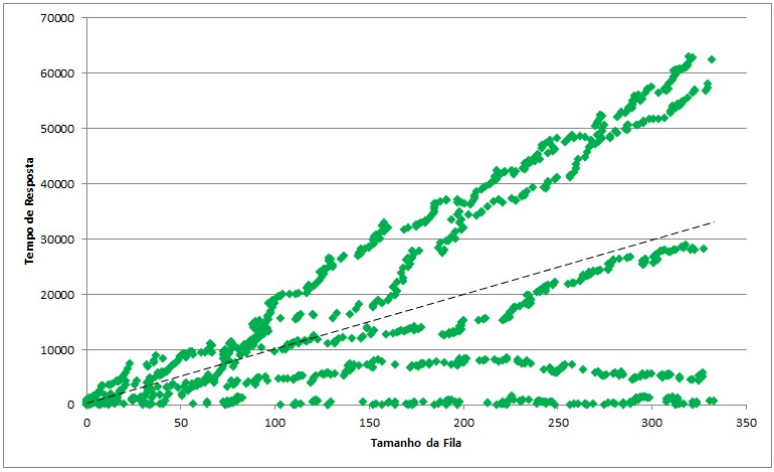

Figura 14: Testes de Implementação - Uma Fila para Todo o Sistema

Tabela 12: Testes de Implementação - Uma Fila para Todo o Sistema

Característica	**Resultados**
Taxa de Perda de Deadline	0,88
Coeficiente de Determinação (R^2)	0,3011
Taxa Relativa de Erro - $E(MBH)$	0,1972
Taxa de Previsões Corretas - $PC(MBH)$	0,8006

O coeficiente de determinação é a medida descritiva da proporção da variação de Y que pode ser explicada por variação em X (BARBETTA; REIS; BORNIA, 2004), ou seja, somente 30,11% do tempo total da thread pode ser explicado pelo tamanho da fila de prontos do processador.

Uma simples alteração da implementação pode fazer com que os resultados melhorem significativamente. Para isto, é necessário que seja armazenado somente a quantidade de eventos pendentes do respectivo *Tratador* e não do total do sistema. Isto significa que será gerado um *Histórico* por *Tratador* e, consequentemente, cada *Tratador* terá sua própria regressão linear. Para comparação, a Figura 15 e a Tabela 13 apresentam o resultado deste teste com as mesmas especificações do teste anterior (detalhadas na Tabela 7).

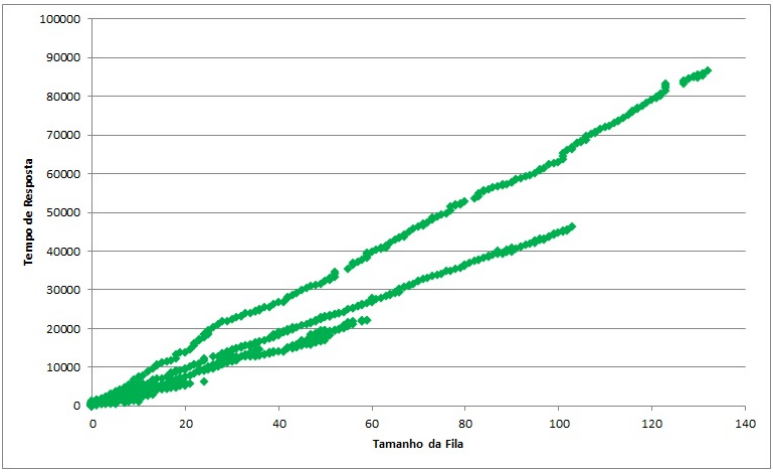

Figura 15: Testes de Implementação - Uma Fila por *Tratador*

Tabela 13: Testes de Implementação - Uma Fila por *Tratador*

Característica	**Resultados**
Taxa de Perda de Deadline	0,91
Coeficiente de Determinação Médio (R^2)	0,9516
Taxa Relativa de Erro - $E(MBH)$	0,0817
Taxa de Previsões Corretas - $PC(MBH)$	0,9216

Como pode ser observado, agora o coeficiente de determinação médio (calculado pela média dos R^2 de cada *Tratador*) demonstra que 95,16% do tempo total da thread pode ser explicado pelo tamanho da

fila de prontos do processador. Isto comprova que estas são duas ótimas variáveis para serem utilizadas pelo mecanismo de previsão, tendo vista o alto grau de relação entre uma e outra. Além disto, também demonstra que quanto mais especializado for o *Histórico*, melhor será a previsão.

Outra característica importante do sistema é o que acontece com a fila de eventos do *Tratador* após a realização do tratamento. Duas abordagens podem ser realizadas: retirar da fila somente aquele evento tratado, ou retirar todos eles. O primeiro é característico de sistemas que necessitam do tratamento de cada um dos eventos, encontrado, por exemplo, em sistemas que realizam operações matemáticas em suas variáveis. O Java RTS trata isto através da utilização do método `getAndDecrementPendingFireCount()` como critério de parada do `handleAsyncEvent()`, implementado pelo *Tratador*.

Já o segundo, é característico de sistemas em que somente uma única informação é importante, a exemplo de sistemas que apenas apresentam/atualizam um dado em um visor. A implementação disto é realizada através do método `getAndClearPendingFireCount()`. Esta abordagem torna o sistema mais aleatório e, consequentemente, mais complexo para a realização de previsões, como demonstram a Figura 16 e a Tabela 14. Entretanto, apesar de deteriorar o coeficiente de determinação médio, o MBH ainda realiza uma ótima previsão.

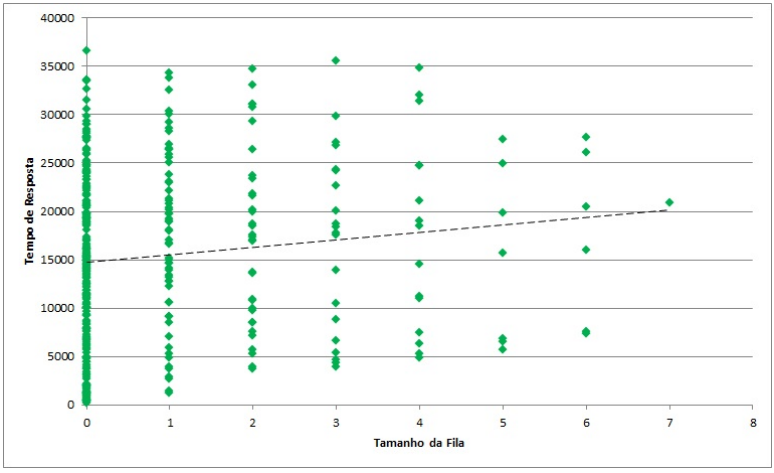

Figura 16: Testes de Implementação - `getAndClearPendingFireCount()`

Tabela 14: Testes de Implementação - `getAndClearPendingFireCount()`

Característica	Resultados
Taxa de Perda de Deadline	0,05
Coeficiente de Determinação Médio (R^2)	0,1056
Taxa Relativa de Erro - $E(MBH)$	0,0257
Taxa de Previsões Corretas - $PC(MBH)$	0,9647

Curiosamente observa-se que o percentual de perda de deadline baixou de 91% para somente 5%. Isto ocorre, pois a cada tratamento de um *Evento*, os demais que estavam na mesma fila deste *Tratador* são considerados como tratados e são descartados sem realizar a sua computação.

5.5 CARACTERÍSTICAS DAS CARGAS DO SISTEMA

Existem diversas interpretações sobre a carga de um sistema. Nesta dissertação a carga é definida pelo percentual de perda de deadline das threads que representam os tratadores de eventos. Uma carga baixa é encontrada quando até 30% das threads perdem seus deadlines, é alta quando mais de 70% das threads perdem seus deadlines e é considerada média quando o percentual de perda de deadline fica entre 30% e 70%.

Inicialmente, uma carga é considerada alta se o seu tempo de computação está muito próximo de seu deadline, fazendo com que o sistema perca a maioria dos seus deadlines, e é considerada baixa se esta diferença for proporcionalmente grande. Entretanto, existem outras variáveis que acabam contribuindo para transformar a carga do sistema. Por exemplo, um sistema cujo tempo de computação das tarefas é muito próximo de seus deadlines ainda pode ter um processador ocioso e, consequentemente, garantir que nenhuma tarefa perca seu deadline.

Com o objetivo de tornar o sistema de testes o mais próximo da realidade e também de abranger uma maior gama de possibilidades, todas as variáveis essenciais (apresentadas na Tabela 6, pág. 65) são aleatórias e influenciam no comportamento do sistema. Assim, nos testes não é possível prever qual será a carga do sistema antes da finalização de sua execução. A carga só pode ser prevista se algumas variáveis se tornam fixas e variam-se poucas das demais. Um exemplo de teste controlado é variar a relação tcT e dT e manter todas as demais

fixas, ou mesmo variar somente o *pG* e deixar as demais fixas.

Normalmente em um sistema de tempo real, esta impossibilidade de previsão da carga não ocorre. Tudo deve ser levado em consideração na fase de projeto do sistema, inclusive a adição de um mecanismo de previsão de perda de deadline.

Os testes demonstraram que aumentar a aleatoriedade dos valores definidos na Tabela 6 não alteram a eficácia dos mecanismos. Entretanto, é possível pensar que a aleatoriedade das variáveis que compõem o sistema colabora para dificultar as previsões.

Para analisar o comportamento dos mecanismos em várias cargas, foram executados testes com as variáveis aleatórias até que o comportamento médio do sistema fosse demonstrado. A partir disto, foram selecionados testes cujos resultados caracterizaram carga baixa, média e alta. Cada um destes três testes foram novamente executados para uma análise mais aprofundada do comportamento dos mecanismos. Todos estes testes são apresentados na sessão a seguir.

5.6 RESULTADOS DOS TESTES

As tabelas e figuras encontradas nesta seção apresentam os resultados dos testes executados com os mecanismos. Estes ambientes realizam o tratamento de todos os eventos do sistema e utilizam uma fila, consequentemente um *Histórico*, por *Tratador*.

São considerados os valores da Tabela 6 (pág. 65) para os testes genéricos, que avaliam o comportamento dos mecanismos através das métricas, e os valores da Tabela 7 (pág. 66) para os testes específicos, que avaliam o comportamento dos mecanismos em uma carga específica do sistema. Neste último caso, somente o valor de *pG* é alterado para caracterizar as diferentes cargas do sistema.

5.6.1 Testes de Carga com o MBF

Estes testes foram realizados em ambiente monoprocessado e multiprocessado. Nesta subseção serão detalhados os resultados do ambiente monoprocessado, pois seus resultados foram melhores. Nas Subseções 5.6.3 e 5.6.4 serão apresentados os comparativos dos resultados nestes dois ambientes.

Lembrando que cada um dos cinco *Tratadores*, definidos na Tabela 7, possui o seu próprio *Histórico* e consequentemente sua própria

regressão linear, as Figuras 17, 18 e 19 apresentam, respectivamente, o comportamento de um teste com carga alta, média e baixa do sistema, utilizando o MBF. As Tabelas 15, 16 e 17 apresentam as características de cada um destes testes.

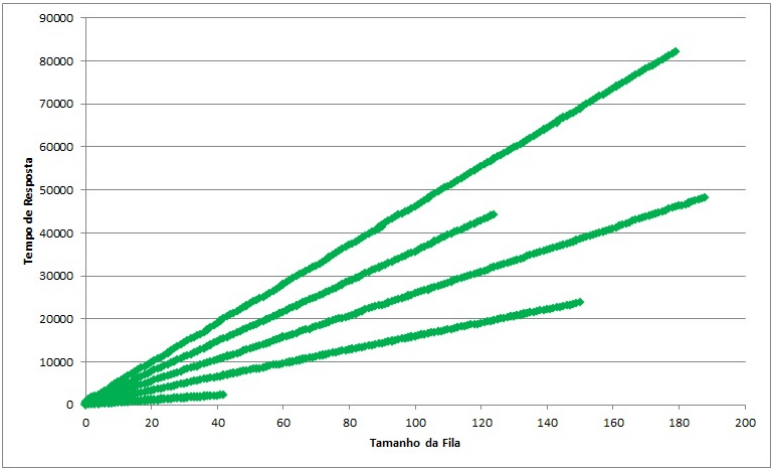

Figura 17: Testes com MBF - Teste com Carga Alta do Sistema

Tabela 15: Testes com MBF - Teste com Carga Alta do Sistema

Característica	**Valores**
Período do Gerador (pG)	100 ms
Taxa de Perda de Deadline	0,95
Taxa Relativa de Erro - $E(MBF)$	0,00
Taxa de Previsões Corretas - $PC(MBF)$	1,00

Tabela 16: Testes com MBF - Teste com Carga Média do Sistema

Característica	**Valores**
Período do Gerador (pG)	1000 ms
Taxa de Perda de Deadline	0,54
Taxa Relativa de Erro - $E(MBF)$	0,0008
Taxa de Previsões Corretas - $PC(MBF)$	0,9991

Cada uma destas figuras foram geradas por uma execução do sistema e, assim como na Seção 5.4, cada ponto representa um *Sinal*

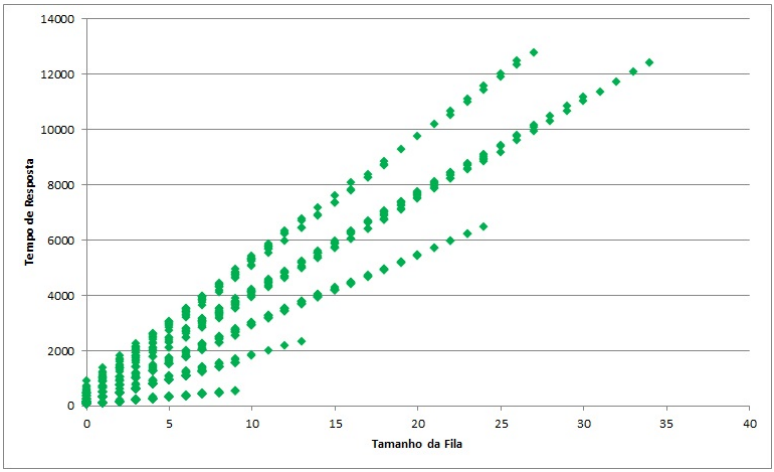

Figura 18: Testes com MBF - Teste com Carga Média do Sistema

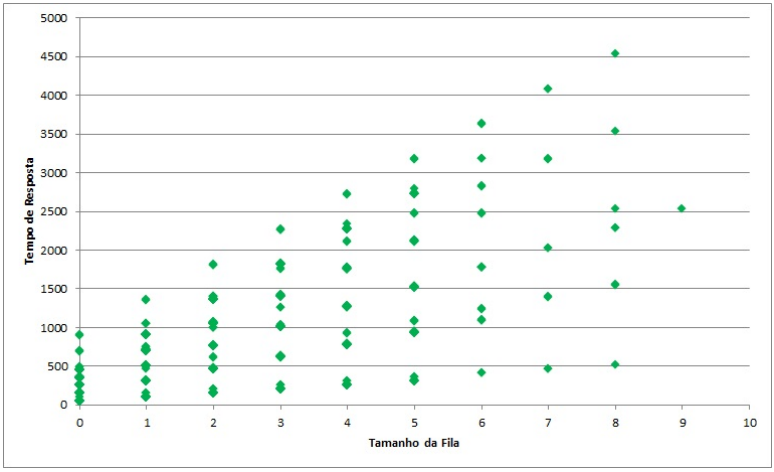

Figura 19: Testes com MBF - Teste com Carga Baixa do Sistema

gerado e a relação entre o tamanho da fila de prontos do processador e o tempo total desde o disparo do evento até a finalização do tratamento.

Pode ser observado claramente nas Figuras 17 e 18 a existência de cinco grupos de sinais com uma mesma tendência, caracterizando os cinco *Tratadores*. O mesmo ocorre na Figura 19, mas como a carga

Tabela 17: Testes com MBF - Teste com Carga Baixa do Sistema

Característica	**Valores**
Período do Gerador (pG)	5000 ms
Taxa de Perda de Deadline	0,12
Taxa Relativa de Erro - $E(MBF)$	0,00
Taxa de Previsões Corretas - $PC(MBF)$	1,00

baixa não gerou grandes filas, não fica tão claro a existências destes grupos de sinais.

Utilizando a Tabela 16 e a Figura 18 como exemplo, a fila chegou a até 35 eventos em um mesmo *Tratador*, acarretando uma taxa de perda de deadline de 54% e caracterizando uma carga média, com eventos levando até 13000 ms para completar a execução, sendo que os tempos de computação variaram entre 50 ms e 450 ms. Com isto, observa-se que a taxa relativa de erro foi de 0,0008, valor bem próximo de 0, e a taxa de previsões corretas com valor muito próximo de 1, comprovando uma ótima previsão.

5.6.2 Testes de Carga com o MBH

Assim como apresentado na subseção anterior, estes testes foram realizados em ambiente monoprocessado e multiprocessado. Entretanto, os resultados foram melhores no ambiente multiprocessado, sendo estes resultados apresentados a seguir. Nas subseções sequentes serão apresentados os comparativos dos resultados nos dois ambientes.

Os valores de pG são similares aos utilizados na subseção anterior. Não são os mesmos, pois como existem mais processadores, a carga do sistema acaba sendo inferior. As Figuras 20, 21 e 22 apresentam, respectivamente, o comportamento de um teste com carga alta, média e baixa do sistema, utilizando o MBH. As Tabelas 18, 19 e 20 apresentam as características de cada um destes testes.

Utilizando a Tabela 19 e a Figura 21 como exemplo, a fila chegou a até 55 eventos em um mesmo *Tratador*, acarretando uma taxa de perda de deadline de 46%, caracterizando uma carga média. Assim como no exemplo da subseção anterior, os tempos de computação variaram entre 50 ms e 450 ms, mas nesta execução houveram eventos que demoraram quase 35000 ms para completar a execução. Com isto, observa-se um coeficiente de determinação de 79,22%, uma taxa relativa de erro de 0,1119, e uma taxa de previsões corretas com valor de

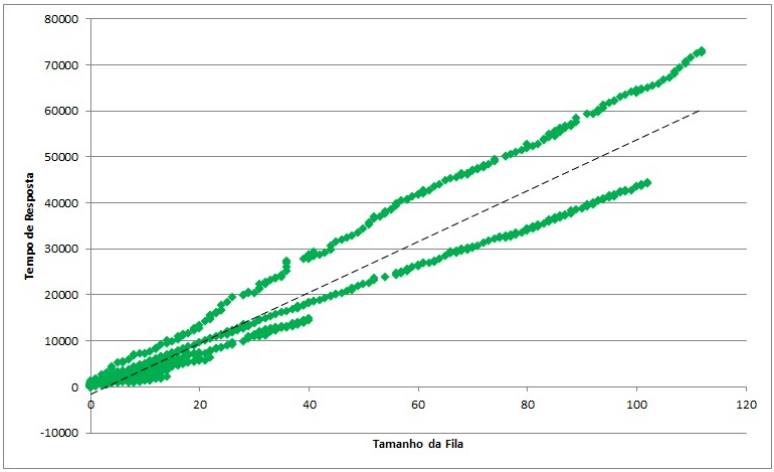

Figura 20: Testes com MBH - Teste com Carga Alta do Sistema

Tabela 18: Testes com MBH - Teste com Carga Alta do Sistema

Característica	**Valores**
Período do Gerador (pG)	500 ms
Taxa de Perda de Deadline	0,92
Coeficiente de Determinação (R^2)	0,9190
Taxa Relativa de Erro - $E(MBH)$	0,0367
Taxa de Previsões Corretas - $PC(MBH)$	0,9714

Tabela 19: Testes com MBH - Teste com Carga Média do Sistema

Característica	**Valores**
Período do Gerador (pG)	1500 ms
Taxa de Perda de Deadline	0.46
Coeficiente de Determinação (R^2)	0,7922
Taxa Relativa de Erro - $E(MBH)$	0,1119
Taxa de Previsões Corretas - $PC(MBH)$	0,8912

0,8912, demonstrando uma boa previsão.

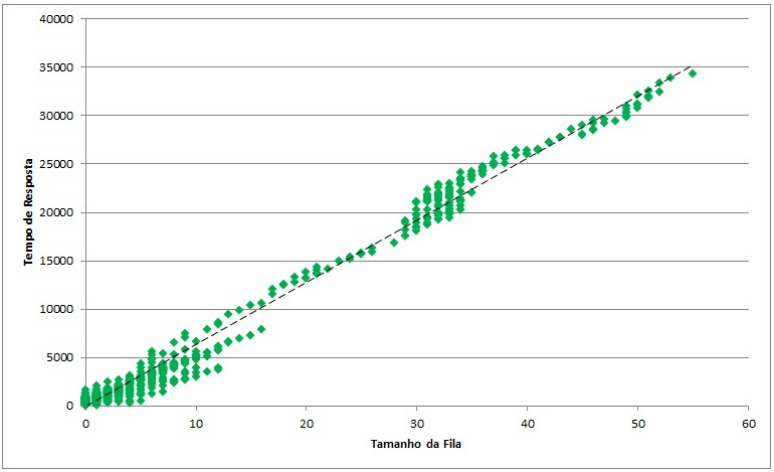

Figura 21: Testes com MBH - Teste com Carga Média do Sistema

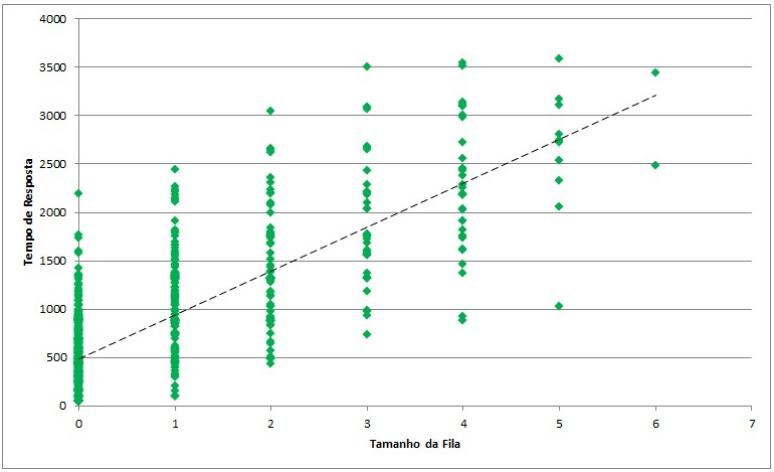

Figura 22: Testes com MBH - Teste com Carga Baixa do Sistema

5.6.3 Testes em Ambiente Monoprocessado

Foram realizados 500 testes em um ambiente monoprocessado, onde ambos os mecanismos realizavam as previsões. A Figura 23 apresenta o gráfico de 500 testes com a relação da Taxa Relativa de Erro

Tabela 20: Testes com MBH - Teste com Carga Baixa do Sistema

Característica	**Valores**
Período do Gerador (pG)	6000 ms
Taxa de Perda de Deadline	0,12
Coeficiente de Determinação (R^2)	0,8920
Taxa Relativa de Erro - $E(MBH)$	0,0368
Taxa de Previsões Corretas - $PC(MBH)$	0,9631

($E(z)$) e do percentual de perda de deadline. Segundo esta métrica, o mecanismo MBF apresenta excelentes resultados (valores próximos de 0) em todas as cargas, mas o mesmo só pode ser observado nas cargas extrema baixa e extrema alta do MBH, apesar do mesmo ainda apresentar uma boa previsão para as demais. O mesmo pode ser observado com a Taxa Previsões Corretas ($PC(z)$) na Figura 24. Neste, o MBF apresenta excelentes resultados (valores próximos a 1) em todas as cargas, mas o mesmo só ocorre nas cargas extremas do MBH.

Nas Tabelas 21 e 22, a coluna *Perda de Deadline* apresenta 10 cargas resultantes do sistema, variando entre 0% e 100% de perda de deadline. Para cada uma das métricas ($E(z)$ e $PC(z)$) é apresentado seu valor médio, o intervalo de confiança de 95% (IC) e, somente para o MBH, o coeficiente de determinação (R^2). Estes valores médios das métricas são apresentados na Figura 25.

Tabela 21: Ambiente Monoprocessado - 500 Testes com MBF - $E(MBF)$ e $PC(MBF)$

Perda de	**E(MBF)**		**PC(MBF)**	
Deadline (%)	Média	IC	Média	IC
0 - 9	0,00033	± 0,00011	0,99967	± 0,00011
10 - 19	0,00184	± 0,00086	0,99816	± 0,00086
20 - 29	0,00134	± 0,00054	0,99866	± 0,00054
30 - 39	0,00272	± 0,00099	0,99728	± 0,00099
40 - 49	0,00242	± 0,00073	0,99758	± 0,00073
50 - 59	0,00196	± 0,00160	0,99804	± 0,00160
60 - 69	0,00202	± 0,00108	0,99798	± 0,00108
70 - 79	0,00133	± 0,00065	0,99867	± 0,00065
80 - 89	0,00105	± 0,00043	0,99895	± 0,00043
90 - 100	0,00116	± 0,00078	0,99884	± 0,00078

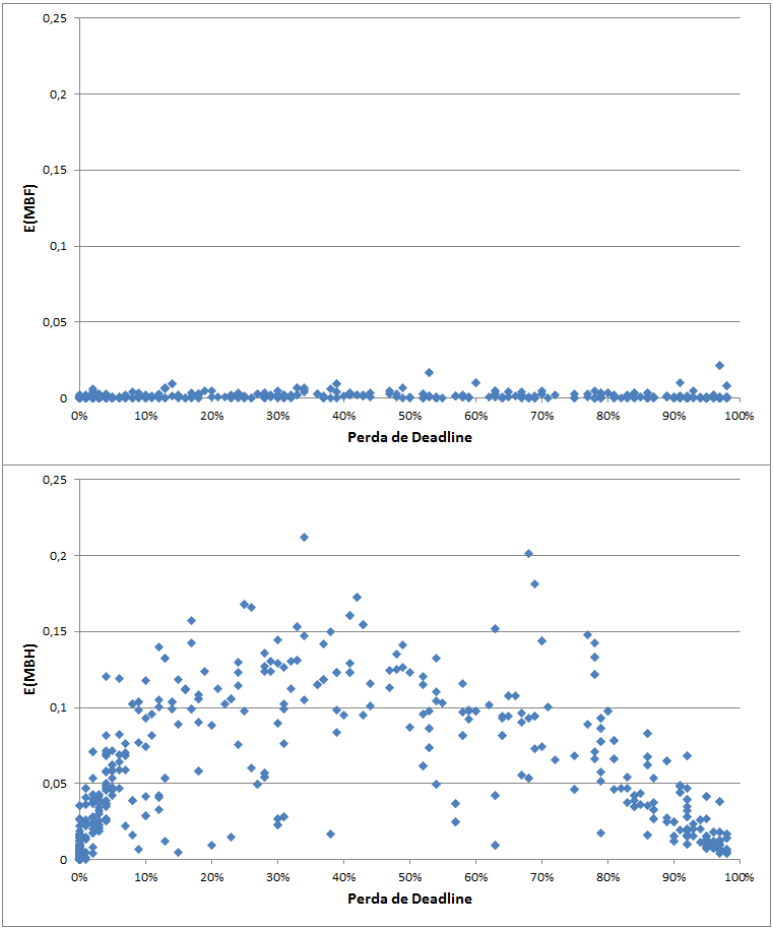

Figura 23: Ambiente Monoprocessado - 500 Testes - $E(z)$

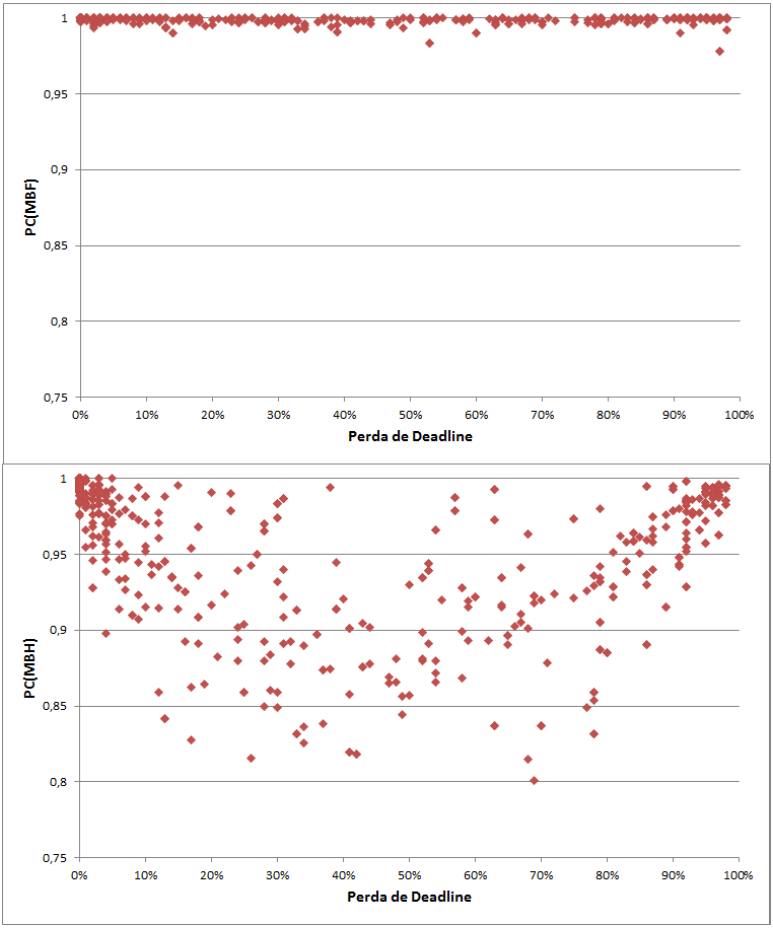

Figura 24: Ambiente Monoprocessado - 500 Testes - $PC(z)$

Tabela 22: Ambiente Monoprocessado - 500 Testes com MBH - $E(MBH)$, $PC(MBH)$ e R^2

Perda de	**E(MBH)**		**PC(MBH)**		$\boldsymbol{R^2}$
Deadline (%)	Média	IC	Média	IC	
0 - 9	0,02052	± 0,00338	0,98681	± 0,00256	0,78274
10 - 19	0,08744	± 0,01254	0,92987	± 0,01425	0,87976
20 - 29	0,09633	± 0,01771	0,91830	± 0,01994	0,93383
30 - 39	0,10318	± 0,01734	0,90724	± 0,01838	0,86383
40 - 49	0,13282	± 0,01207	0,86704	± 0,01583	0,95932
50 - 59	0,08347	± 0,01147	0,92164	± 0,01498	0,91830
60 - 69	0,09150	± 0,01728	0,91406	± 0,01906	0,94030
70 - 79	0,09066	± 0,01473	0,90173	± 0,01895	0,92401
80 - 89	0,05070	± 0,00732	0,94485	± 0,00953	0,93808
90 - 100	0,01962	± 0,00306	0,98000	± 0,00354	0,98114

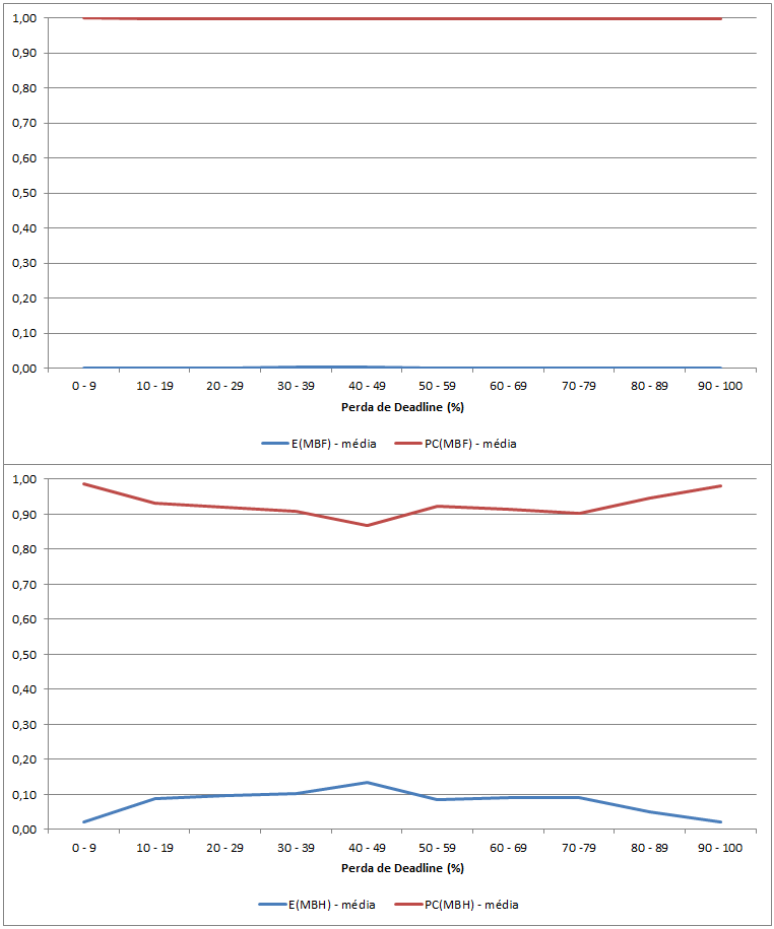

Figura 25: Ambiente Monoprocessado - 500 Testes - Valor Médio de $E(z)$ e $PC(z)$

5.6.4 Testes em Ambiente Multiprocessado

Foram realizados 500 testes em um ambiente multiprocessado, onde ambos os mecanismos realizavam as previsões. A Figura 26 apresenta o gráfico de 500 testes com a relação da Taxa Relativa de Erro (*E(z)*) e do percentual de perda de deadline. Observa-se em ambos os mecanismos que nas cargas baixas e altas (lateral esquerda e direita da figura) a métrica apresenta ótimos resultados (valor tendendo a 0), mas estes resultados não são tão significativos na carga média do sistema. O mesmo pode ser observado com a Taxa Previsões Corretas (*PC(z)*) na Figura 27. Nestes, as cargas altas e baixas apresentam resultados esperados (valor tendendo a 1), mas somente bons resultados na carga média do sistema. É possível pensar que a aleatoriedade das variáveis que compõem o sistema colabora para dificultar essa previsão.

Nas Tabelas 23 e 24 a coluna *Perda de Deadline* apresenta 10 cargas resultantes do sistema, variando entre 0% e 100% de perda de deadline. Para cada uma das métricas (*E(z)* e *PC(z)*) é apresentado seu valor médio, o intervalo de confiança de 95% (IC) e, somente para o MBH, o coeficiente de determinação (R^2). Os valores médios destas métricas são apresentados na Figura 28.

Tabela 23: Ambiente Multiprocessado - 500 Testes com MBF - $E(MBF)$, $PC(MBF)$

Perda de	**E(MBF)**		**PC(MBF)**	
Deadline (%)	Média	IC	Média	IC
0 - 9	0,03037	± 0,00613	0,97837	± 0,00485
10 - 19	0,09179	± 0,01055	0,91996	± 0,01029
20 - 29	0,13219	± 0,00935	0,88372	± 0,01029
30 - 39	0,14442	± 0,00874	0,87287	± 0,01020
40 - 49	0,12590	± 0,01489	0,89098	± 0,01511
50 - 59	0,11732	± 0,00979	0,89995	± 0,01051
60 - 69	0,09689	± 0,00719	0,91464	± 0,00790
70 - 79	0,08502	± 0,01059	0,92556	± 0,01151
80 - 89	0,05411	± 0,00909	0,95177	± 0,00996
90 - 100	0,02260	± 0,00360	0,97742	± 0,00383

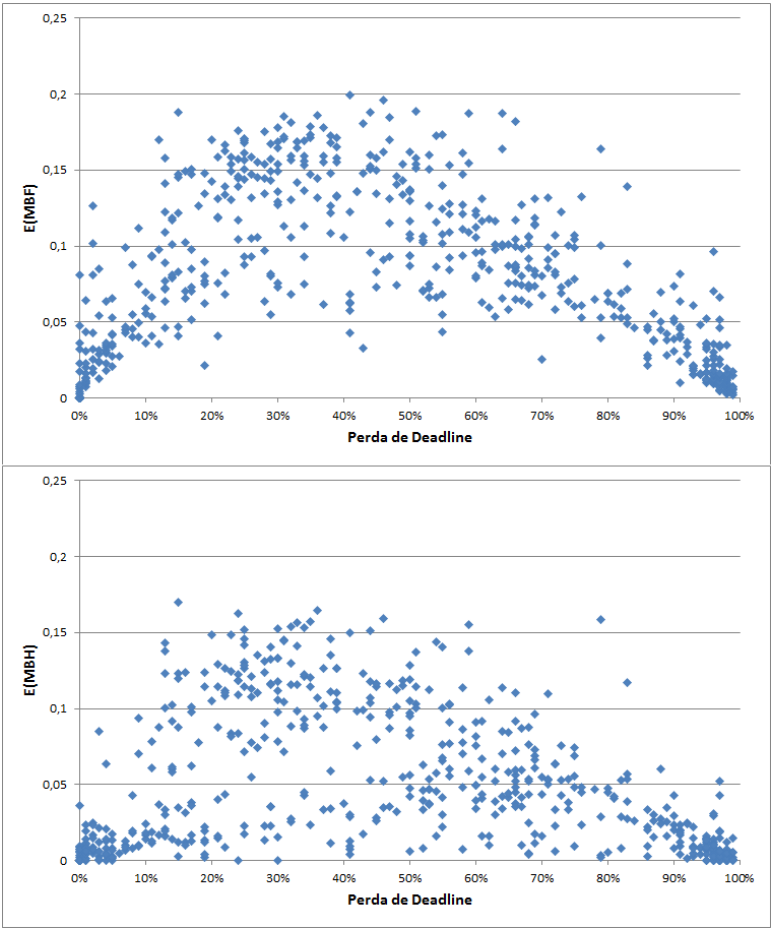

Figura 26: Ambiente Multiprocessado - 500 Testes - $E(z)$

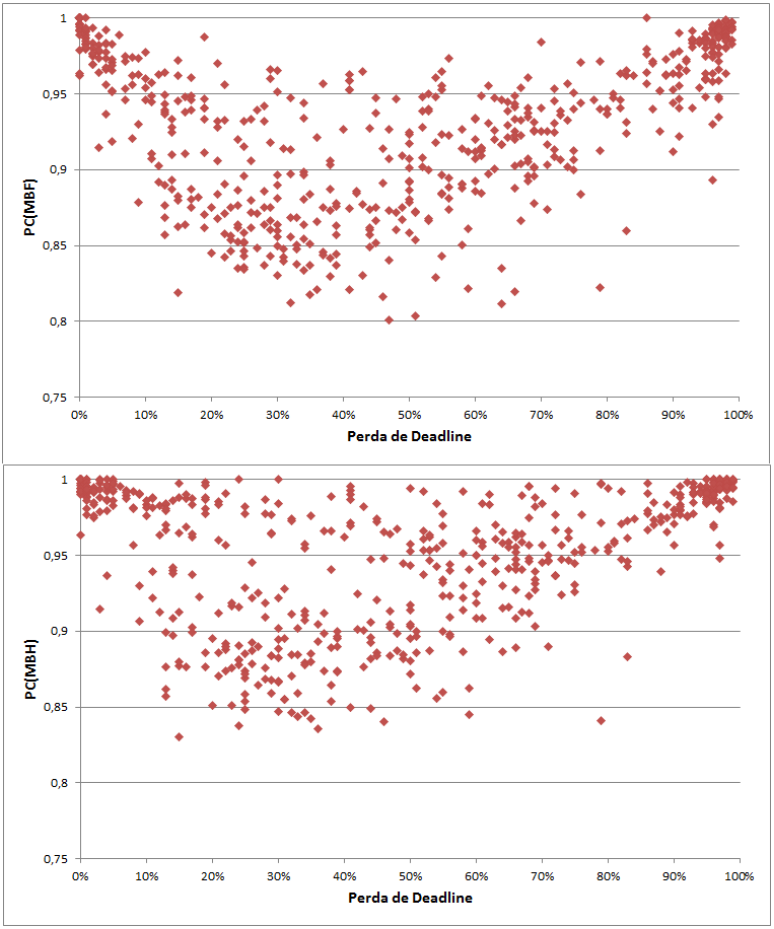

Figura 27: Ambiente Multiprocessado - 500 Testes - $PC(z)$

Tabela 24: Ambiente Multiprocessado - 500 Testes com MBH - $E(MBH)$, $PC(MBH)$ e R^2

Perda de	**E(MBH)**		**PC(MBH)**		$\boldsymbol{R^2}$
Deadline (%)	Média	IC	Média	IC	
0 - 9	0,01074	± 0,00369	0,98926	± 0,00369	0,78290
10 - 19	0,05283	± 0,01244	0,94717	± 0,01244	0,71497
20 - 29	0,09335	± 0,01217	0,90665	± 0,01217	0,70434
30 - 39	0,10033	± 0,01154	0,89967	± 0,01154	0,64871
40 - 49	0,07633	± 0,01504	0,92367	± 0,01504	0,69507
50 - 59	0,07353	± 0,01060	0,92647	± 0,01060	0,72305
60 - 69	0,05440	± 0,00711	0,94560	± 0,00711	0,78668
70 - 79	0,04821	± 0,01131	0,95179	± 0,01131	0,81660
80 - 89	0,03354	± 0,00894	0,96646	± 0,00894	0,87649
90 - 100	0,00889	± 0,00190	0,99111	± 0,00190	0,97096

Figura 28: Ambiente Multiprocessado - 500 Testes - Valor Médio de $E(z)$ e $PC(z)$

5.7 CONCLUSÕES

Neste capítulo inicialmente foi apresentado um estudo de caso do *cruise control*, sistema de tempo real firme composto por vários sensores e atuadores que caracterizam tarefas aperiódicas. A partir da generalização deste estudo de caso, foi especificado um ambiente de teste monoprocessado, e outro multiprocessado, que utilizam Java RTS em um computador de propósito geral, onde outros processos de controle do sistema operacional são executados, rodando em um Ubuntu com kernel de tempo real.

Foram realizados testes de cargas específicas (baixa, média e alta) com ambos os mecanismos de previsão em ambos os sistemas. Nestes testes as variáveis de controle foram especificadas numericamente para caracterizar cada uma destas cargas. Também foram realizados testes em que estas variáveis eram aleatórias, para conseguir abranger uma ampla gama de possibilidades de características do sistema.

Se um sistema de tempo real se comporta em carga baixa, há um desperdício de recursos. Se o sistema trabalha sempre em carga alta, houve algum erro de projeto, já que a maioria das tarefas sempre estará perdendo seu deadline. Desta forma, espera-se que um sistema de tempo real normalmente se comporte com uma carga média. E é este o cenário mais complexo para a realização de previsões. Como regra geral, quanto mais variável o comportamento do sistema, mais difícil é para fazer qualquer previsão sobre o cumprimento de deadlines (PLENTZ; MONTEZ; OLIVEIRA, 2011b).

Embora não seja evidente quando se observa os resultados agrupados de todos os testes (Figuras 26 e 27), a Figura 28 mostra que ambos mecanismos se comportam bem quando o sistema está com carga baixa, média e alta. Comparando os valores médios das métricas apresentadas nas Tabelas 23 e 24, considerando os intervalos de confiança, é possível observar que o MBH se comporta melhor que o MBF no ambiente multiprocessado, como também pode ser observado na Figura 28.

Retornando à Figura 25 e comparando com a Figura 28 é possível observar que o MBH mantêm o mesmo padrão de eficácia, mas o mesmo não ocorre com o MBF. Isto, e outros testes realizados, demonstram que a medida que a complexidade do sistema aumenta, a eficácia do MBF diminui e a do MBH se mantêm estável.

É possível dizer que os mecanismos mostram muito bons resultados em baixas e altas cargas porque é mais fácil de fazer previsões nestes sistemas. Cargas baixas possuem amplo tempo disponível para

completar seus deadlines e cargas altas tornam difícil cumprir os deadlines de qualquer maneira. As cargas médias refletem a maioria dos sistemas de tempo real, em que há tempo suficiente para completar o deadline se o sistema permanece estável na maior parte do tempo. Este tipo de sistema é de interesse fundamental e os mecanismos propostos apresentam resultados satisfatórios.

6 CONCLUSÕES E TRABALHOS FUTUROS

Através de um adequado mecanismo de previsão de perda de deadline é possível tomar medidas corretivas, tais como eliminar a tarefa ou aumentar a sua prioridade, para aprimorar o desempenho do sistema. Este tipo de mecanismo permite a avaliação de novos projetos de sistemas de tempo real, caso a quantidade de tarefas que perdem o deadline não estiver dentro dos parâmetros definidos, ou mesmo avaliar sistemas que já estão em funcionamento e que sofreram alterações na sua composição, como a adição ou retirada de recursos computacionais. A contribuição principal desta dissertação é a definição de dois previsores de perda de deadline para sistemas monoprocessados e multiprocessados de tempo real firme que trazem bons resultados em ambientes com cargas baixas, médias e altas. Além disto, os resultados preliminares deste trabalho foram publicados em (JUNIOR; PLENTZ, 2013).

Foram pesquisadas algumas informações que poderiam compor o histórico e, devido ao alto valor do coeficiente de determinação, foi definido que seria composto pelo tamanho da fila de pronto do processador e pelo tempo de resposta da thread em execuções passadas. Foi desenvolvido um modelo de tarefas para aplicações não críticas em um sistema de tempo real firme, caracterizando uma aplicação real, o *cruise control*. Optou-se por fazer os testes com várias configurações, caracterizando um ambiente generalista, porque considera-se mais interessantes manter e avançar os testes em um ambiente com características que refletem um ambiente real, considerando que o desafio é trazer a teoria e conseguir os resultados necessários de tempo real em um ambiente comum de aplicações convencionais.

O Mecanismo de Previsão de Perda de Deadline Baseado na Folga (MBF) utiliza algumas informações (deadline, tempo de computação e o tempo de espera na fila de prontos do *Tratador*) para calcular a folga e realizar a previsão. Este mecanismo apresentou excelentes resultados em sistemas monoprocessados. Utilizando regressão linear, o Mecanismo de Previsão de Perda de Deadline Baseado no Histórico (MBH) relaciona as informações do histórico com o tamanho da fila de prontos do *Tratador* para calcular o tempo de resposta previsto da thread. Este tempo de resposta é utilizado para definir a probabilidade da thread cumprir o seu deadline. O MBH apresentou ótimos resultados em sistemas multiprocessados.

Após os testes, foram avaliadas a qualidade das previsões utili-

zando as Métricas Taxa Relativa de Erro ($E(z)$) e Taxa de Previsões Corretas ($PC(z)$). Como em qualquer análise, as conclusões são válidas principalmente para cenários de aplicação semelhantes aos testados. Com este intuito, os testes foram feitos utilizando uma implementação em Java RTS em um ambiente não especialista, próximo de um ambiente de tempo real comum (CC), onde diversas decisões e variáveis eram aleatórias, abrangendo uma grande gama de cenários.

Retomando os trabalhos relacionados, a Tabela 25 apresenta um comparativo entre eles e os dois mecanismos propostos. Nela é possível observar que a maioria dos demais trabalhos validaram os mecanismos através de simulações, onde há um maior controle dos resultados esperados, enquanto que os dois mecanismos propostos foram validados através de implementação e medição de fato, onde várias peculiaridades impactaram nos resultados. Além disto, ambos os mecanismos propostos foram implementados utilizando uma linguagem de programação para tempo real e acarretam pouco overhead sobre o sistema, quando comparado com os demais.

Dado o formato que estes mecanismos foram desenvolvidos, implementados e testados, não precisam ficar restritos aos tratadores de eventos e podem ser utilizados em qualquer sistema que possua tarefas aperiódicas de tempo real, desde que estas tarefas possuam deadline e tempo de computação. Desenvolver um mecanismo de previsão de perda de deadline para tratadores de eventos do RTSJ foi interessante porque mostrou a possibilidade de redefini-lo como um componente caixa-preta de software, que pode ser aplicado em qualquer tipo de sistema de tempo real firme. Esta pesquisa abre uma nova possibilidade de implementar um módulo de previsão de perda de deadline como um importante recurso para sistema em tempo real firmes e não críticos.

Para pesquisas e trabalhos futuros, um assunto que se demonstrou interessante é o aumento da complexidade do sistema, alterando, por exemplo, a proporção de *Tratadores* por *Evento*, fazendo com que um *Evento* possa ser tratado por mais de um *Tratador*, ou mesmo que um *Tratador* seja responsável por mais de um *Evento*. Outra possibilidade de aumento da complexidade é a utilização de tarefas com prioridade dinâmica. Seria interessante verificar o comportamento dos mecanismos com essas variações, pois elas refletem características encontradas em sistemas de tempo real que podem levar a modificações nos mecanismos, trazendo novas informações a serem incorporadas na previsão.

Em muitos sistemas de tempo real novos, ou mesmo em sistemas que já estão sendo executados, não existem espaço, tempo ou recursos

para a realização de uma fase de levantamento de informações para o *Histórico* utilizado pelo MBH. Neste tipo de sistema, uma possível abordagem é utilizar uma mescla entre os dois mecanismos propostos, gerando um terceiro. Na fase de *Inicialização* este mecanismo seguiria o algoritmo de utilização do MBH com a adição do cálculo de previsão de perda de deadline do MBF, e nas fases de *Execução* e *Finalização* ele seguiria conforme definido pelo algoritmo de utilização do MBH, inclusive no seu modo iterativo. Faz-se necessário testes para comparar a eficácia desse mecanismo com os dois mecanismos propostos, verificando seu comportamento em todas as possíveis cargas, mas há indicações de que esta é uma interessante abordagem para estes sistemas.

Os resultados obtidos demonstram a importância dos mecanismos de previsão. Não foi objeto de análise desta dissertação, mas poderá ser trabalhado futuramente, tanto a tomada de decisão após a previsão de que uma tarefa irá perder seu deadline, quanto a utilização de técnicas de inteligência artificial para modelar um mecanismo de previsão, considerando as características dos mecanismos e do sistema apresentados.

Tabela 25: Comparativo entre os Trabalhos Correlatos, MBF e MBH

Trabalho	**Tipo de Sistema**	**Técnica Aplicada no Mecanismo**	**Sistema de Tempo Real?**	**Utiliza Histórico?**	**Tipo de Validação**	**Linguagem de Programação**	**Overhead**
Feng, Yingying e Nianbo (2009)	Intelligent Transportation System	Rede Neural com *Backpropagation*	Não	Sim	Estudo de Caso	Não identificado	Alto
Prodan e Nae (2009)	MMOGs em Grids	Rede Neural	Sim	Sim	Simulação	C++	Alto
Perrone et al. (2009)	Sistema baseado em Componentes COTS	Distribuição de Probabilidade	Sim	Sim	Simulação	C++	Médio
Tatibana, Montez e Oliveira (2007)	Sistema Embutido	Função de Massa de Probabilidade	Sim	Sim	Simulação	Java	Médio
Plentz, Montez e Oliveira (2008b)	Sistema Distribuído	Regressão Linear	Sim	Sim	Simulação	Java	Médio
Plentz, Montez e Oliveira (2011a)	Sistema Distribuído	Cálculo Matemático	Sim	Não	Simulação	Java	Baixo
Plentz, Montez e Oliveira (2011b)	Sistema Distribuído	Cálculo Matemático	Sim	Não	Simulação	Java	Muito Baixo
Lorbieski, Plentz e Friedrich (2012)	Sistema Distribuído	Regressão Linear	Sim	Sim	Simulação	Java RTS	Médio
MBF	Sistema Monoprocessado	Cálculo Matemático	Sim	Não	Testes	Java RTS	Muito Baixo
MBH	Sistema Multiprocessado	Regressão Linear	Sim	Sim	Testes	Java RTS	Baixo

REFERÊNCIAS

BARBETTA, P. A.; REIS, M. M.; BORNIA, A. C. *Estatística para Cursos de Engenharia e Informática*. [S.l.]: Atlas, 2004. ISBN 9788522459940.

BOYSE, J. W.; WARN, D. R. A straightforward model for computer performance prediction. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 7, n. 2, p. 73–93, jun. 1975. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/356648.356649>.

BROWN, G. E.; ECKHOUSE JR., R.; ESTABROOK, J. Operating system enhancement through firmware. In: *Proceedings of the 10th Annual Workshop on Microprogramming*. Piscataway, NJ, USA: IEEE Press, 1977. (MICRO 10), p. 119–133. Disponível em: <http://dl.acm.org/citation.cfm?id=800102.803324>.

BRUNO, E. J.; BOLLELLA, G. *Real-Time Java Programming: With Java RTS*. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2009. ISBN 0137142986, 9780137142989.

BRYAN, G. E.; SHEMER, J. E. The uts time-sharing system: Performance analysis and instrumentation. In: *Proceedings of the Second Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 1969. (SOSP '69), p. 147–158. Disponível em: <http://doi.acm.org/10.1145/961053.961102>.

BUTTAZZO, G. C. *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*. 3rd. ed. [S.l.]: Springer, 2004.

CRUZ, G. M. aes; LIMA, G. M. de A. Simulador de escalonamento para sistemas de tempo real. In: *IV WTICG - ERBASE*. [S.l.: s.n.], 2006.

DEVARAKONDA, M. V.; IYER, R. K. Predictability of process resource usage: A measurement-based study on unix. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 15, n. 12, p. 1579–1586, dez. 1989. ISSN 0098-5589. Disponível em: <http://dx.doi.org/10.1109/32.58769>.

DEVARAKONDA, M. V. S. *File usage analysis and process resource usage prediction: A measurement based study*. Tese (Tese de Doutorado) — Univ. Illinois at Urbana-Champaign, 1988.

DIBBLE, P. C. *Real-Time Java Platform Programming*. [S.l.]: Sun Microsystems Press Series, 2002. ISBN 9780130282613.

DINDA, P. Online prediction of the running time of tasks. In: *Cluster Computing*. [S.l.: s.n.], 2001. p. 383–394.

DOWNEY, A. B. *Predicting Queue Times on Space-Sharing Parallel Computers*. Berkeley, CA, USA, 1996.

FARINES, J.-M.; FRAGA, J. da S.; OLIVEIRA, R. S. de. *Sistemas de Tempo Real*. SÃ£o Paulo: Escola de Computação, 2000.

FENG, L.; YINGYING, D.; NIANBO, Z. A practical route guidance approach based on historical and real-time traffic effects. In: *Geoinformatics, 2009 17th International Conference on*. [S.l.: s.n.], 2009. p. 1–6.

GOEL, A. L.; OKUMOTO, K. An analysis of recurrent software errors in a real-time control system. In: *Proceedings of the 1978 Annual Conference*. New York, NY, USA: ACM, 1978. (ACM '78), p. 496–501. ISBN 0-89791-000-1. Disponível em: <http://doi.acm.org/10.1145/800127.804160>.

GOSWAMI, K.; IYER, R.; DEVARAKONDA, M. Load sharing based on task resource prediction. In: *System Sciences, 1989. Vol.II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*. [S.l.: s.n.], 1989. v. 2, p. 921–927 vol.2.

HAMMER, M.; CHAN, A. Index selection in a self-adaptive data base management system. In: *Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1976. (SIGMOD '76), p. 1–8. Disponível em: <http://dl.acm.org/citation.cfm?id=509383.509385>.

JCM, J. C. P. *Real-Time Specification for Java - RTSJ*. 2012. Accessed Out., 2013. Disponível em: <http://www.rtsj.org/>.

JUNIOR, R. B.; PLENTZ, P. D. M. *PrevisÃ£o de Perda de Deadline em Tratadores de Eventos RTSJ*. [S.l.: s.n.], 2013.

LAPLANTE, P. *REAL-TIME SYSTEMS DESIGN & ANALYSIS 3rd Ed.* [S.l.]: Wiley India Pvt. Limited, 2006. ISBN 9788126508303.

LITKE, A.; TSERPES, K.; VARVARIGOU, T. Computational workload prediction for grid oriented industrial applications: the case

of 3d-image rendering. In: *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*. [S.l.: s.n.], 2005. v. 2, p. 962–969 Vol. 2.

LIU, J. W. S. *Real-Time Systems*. [S.l.]: Prentice Hall, 2000. ISBN 9780130996510.

LORBIESKI, R.; PLENTZ, P.; FRIEDRICH, L. Implementing distributed threads using rtsj. In: *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on*. [S.l.: s.n.], 2012. p. 265–269.

MAK, V.; LUNDSTROM, S. F. Predicting performance of parallel computations. *Parallel and Distributed Systems, IEEE Transactions on*, v. 1, n. 3, p. 257–270, Jul 1990. ISSN 1045-9219.

MARTIN, J. *Design of Real-Time Computer Systems*. Upper Saddle River, NJ, USA: Prentice Hall, Inc., 1967.

MOORE, G. E. Cramming more components onto integrated circuits. In: . [S.l.: s.n.], 1965.

OLIVEIRA, R. S. de. *Escalonamento de Tarefas Imprecisas em Ambiente Distribuído*. Tese (Tese de Pós-Doutorado) — Programa de Pós-Graduação em Engenharia Elétrica (PPGEEL), Universidade Federal de Santa Catarina, 1997.

PERRONE, R. et al. An approach for estimating execution time probability distributions of component-based real-time systems. In: *Journal of Universal Computer Science, vol. 15, no. 11 (2009)*. [S.l.: s.n.], 2009. p. 2142–2165.

PLENTZ, P. D. M. *Mecanismos de Previsão de Perda de Deadline para Sistemas Baseados em Threads Distribuídas Tempo Real*. Tese (Tese de Doutorado) — Universidade Federal de Santa Catarina, 2008.

PLENTZ, P. D. M. et al. Expressividade da rtsj para implementar computação imprecisa. In: *6o Workshop de Tempo Real, 2004. WTR.* [S.l.: s.n.], 2004.

PLENTZ, P. D. M.; MONTEZ, C.; OLIVEIRA, R. de. Deadline missing prediction in systems based on distributed threads. In: *Robotic Symposium, 2008. LARS '08. IEEE Latin American*. [S.l.: s.n.], 2008. p. 190–195.

PLENTZ, P. D. M.; MONTEZ, C.; OLIVEIRA, R. S. de. Deadline missing predictor based on aperiodic server queue length for distributed systems. *Computer Communications*, v. 31, n. 17, p. 4167–4175, 2008.

PLENTZ, P. D. M.; MONTEZ, C.; OLIVEIRA, R. S. de. As prediction mechanism for distributed threads systems. *Journal of Parallel and Distributed Computing*, v. 71, n. 10, p. 1367–1376, 2011.

PLENTZ, P. D. M.; MONTEZ, C.; OLIVEIRA, R. S. de. Deadline missing prediction through the use of milestones. *Computing and Informatics*, p. 657–679, 2011.

PROCESS, J. C. *JSR 001: Real-time Specification for Java*. 2012. Accessed Out., 2013. Disponível em: <http://jcp.org/en/jsr/detail?id=001>.

PROCESS, J. C. *JSR 282: RTSJ version 1.1*. 2012. Accessed Out., 2013. Disponível em: <http://jcp.org/en/jsr/detail?id=282>.

PRODAN, R.; NAE, V. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Generation Computer Systems*, v. 25, n. 7, p. 785 – 793, 2009. ISSN 0167-739X.

ROSENBLATT, J. R. On prediction of system performance from information on component performance. In: *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*. New York, NY, USA: ACM, 1957. (IRE-AIEE-ACM '57 (Western)), p. 85–94. Disponível em: <http://doi.acm.org/10.1145/1455567.1455583>.

SCHOPF, J. *Structural Prediction Models for High-Performance Distributed Applications*. 1997.

SCHOPF, J. M.; BERMAN, F. Performance prediction in production environments. In: . [S.l.: s.n.], 1998.

SHAW, A. *Sistemas E Software de Tempo Real*. [S.l.]: BOOKMAN COMPANHIA ED, 2001. ISBN 9788536301723.

SMITH, W.; FOSTER, I.; TAYLOR, V. Predicting application run times with historical information. *J. Parallel Distrib. Comput.*, Academic Press, Inc., Orlando, FL, USA, v. 64, n. 9, p. 1007–1016, set. 2004. ISSN 0743-7315. Disponível em: <http://dx.doi.org/10.1016/j.jpdc.2004.06.008>.

TATIBANA, C.; MONTEZ, C.; OLIVEIRA, R. Soft real-time task response time prediction in dynamic embedded systems. In: OBERMAISSER, R. et al. (Ed.). *Software Technologies for Embedded and Ubiquitous Systems*. [S.l.]: Springer Berlin Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4761). p. 273–282. ISBN 978-3-540-75663-7.

WELLINGS, A. *Concurrent and Real-Time Programming in Java*. Wiley, 2005. ISBN 9780470011270. Disponível em: <http://books.google.com.br/books?id=tJ03VlKJacMC>.

WILLIAMS, R. *Real-Time Systems Development*. Elsevier Science, 2005. ISBN 9780080456409. Disponível em: <http://books.google.com.br/books?id=Qi_Eo8Q_k5AC>.

WOLSKI, R.; SPRING, N.; HAYES, J. Predicting the cpu availability of time-shared unix systems on the computational grid. In: *In Proceedings of the Eighth IEEE Symposium on High Performance Distributed Computing HPDC99*. [S.l.: s.n.], 1998. p. 105–112.